Contents

[Project Guidelines 1](#_Toc439042747)

[Global 1](#_Toc439042748)

[Default Rules 2](#_Toc439042749)

[Variable Names 2](#_Toc439042750)

[Method Names 3](#_Toc439042751)

[Class Names 4](#_Toc439042752)

[Interface Names 4](#_Toc439042753)

[Package Name 4](#_Toc439042754)

[File Names 4](#_Toc439042755)

[C# code style guid 4](#_Toc439042756)

[C# Naming 4](#_Toc439042757)

[C# Formatting 5](#_Toc439042758)

[Client Application 6](#_Toc439042759)

[JavaScript 6](#_Toc439042760)

[AngularJS 8](#_Toc439042761)

[Module 9](#_Toc439042762)

[Controllers 11](#_Toc439042763)

[Services 14](#_Toc439042764)

[Directives 15](#_Toc439042765)

[Typescript 15](#_Toc439042766)

[HTML 15](#_Toc439042767)

[CSS 16](#_Toc439042768)

[Latin abbreviations 18](#_Toc439042769)

[Structure 20](#_Toc439042770)

[Explanation 21](#_Toc439042771)

[Documentation of Project: 22](#_Toc439042772)

[References 22](#_Toc439042773)

# Project Guidelines

## Global

##### Length:

* 1 char for loop counters,
* 1 word for condition/loop variables,
* 1-2 words for methods,
* 2-3 words for classes,
* 3-4 words for globals.

##### Characters consistently

Don't use lowercase/uppercase characters inconsistently: e.g. userName, UserName, USER\_NAME, m\_userName, username, ...:

* use Camel Case (aka Upper Camel Case) for classes: VelocityResponseWriter
* use Lower Case for packages: com.company.project.ui
* use Mixed Case (aka Lower Camel Case) for variables: studentName
* use Upper Case for constants : MAX\_PARAMETER\_COUNT = 100
* use Camel Case for enum class names and Upper Case for enum values.
* don't use '\_' anywhere except constants and enum values (which are constants).

##### Reuse variable

* Don't reuse same variable name in the same class in different contexts: e.g. in method, constructor and class.
* Don't use same variable for different purposes in a method, conditional etc.

# Default Rules

**--- For any programming language ---**

## Variable Names

* Use camel case in variable names as well e.g. price, quantity, totalAmount etc.
* Use all caps for constants e.g. MAX\_QUANTITY, MAX\_PRICE etc.
* Avoid using non ASCII characters and words from local language! Using any character other than ASCII character, especially non English is worst practice. English is a universal language for programming and stick with it.
* All names start with a **letter.**
* If you have to represent a collection or plural, prefer something like listOfEmployees, bunchOfEmployees over employees.

##### Avoid Pointless Names:

No: abc, temp, data

##### Avoid Similar Names:

No: employee and employees

##### Avoid Clutters:

**No:** \_, m\_, o\_, simply \_, obj\_

##### Avoid Hungarian notation:

No: bExit for boolean variable, iMax for integer variables

Make good use of common verb e.g. is, has, can or do  
  
Yes: isAlive(), hasNext(), canExecute()…

Example:

if(isRaining){

bringUmbrella();

}

##### Use specific names for variables:

No: "value", "equals", "data"…

Follow Classical Programming Convention:  
  
i and j as loop counter in for loop

E.g.: for(int i=0; i<10; i++){ // your code }

##### Spaces Around Operators

Always put spaces around operators ( = + - \* / )

Yes: Integer x = 42; var x = y + z; var values = ["Volvo", "Saab", "Fiat"];  
No: Integer x=42;

## Method Names

* Start name of method from small character and follow camel case e.g. getEmployee(), getPayDate() etc.
* Instead of having three different methods destroy(), kill(), or finish() at different modules, prefer one of them e.g. destroy().
* Method names should start with verb e.g. get, set, do, invoke etc.
* Prefer shorter name over longer one, if it reveal intent clearly: getPayDate() is better than retreivePaymentDate().
* Prefer shorter name if and only if it reveal intent completely, otherwise choose longer and descriptive name: getLiquidityIndicator() is better than getLInd().
* If variable name is payDate then getter method name must be getPayDate()and setter method must be setPayDate().

##### Give Meaningful Names:

Yes: getPayDate()

No: getPD()

## Class Names

Start name of class as capital letter e.g. Employee, Student or Thread.  
  
Class name should be noun and should tell what does this class represent e.g. Employee, Thread, String etc.

## Interface Names

Interface name should describe ability or CAN DO part e.g.

[Runnable - can run](http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html), Callable - can be called etc. 

## Package Name

Package name should follow standard company structure e.g. com.company.project.module.

## File Names

Always use lower case file names (if possible).

# C# code style guides

## C# Naming

1. Always use Camel Case or Pascal Case names.
2. Avoid ALL CAPS and all lowercase names. Single lowercase words or letters are acceptable.
3. Do not create declarations of the same type (namespace, class, method, property, field, or parameter) and access modifier (protected, public, private, internal) that vary only by capitalization.
4. Do not use names that begin with a numeric character.
5. Do add numeric suffixes to identifier names.
6. Always choose meaningful and specific names.
7. Variables and Properties should describe an entity not the type or size.
8. Do not use Hungarian Notation!
9. Avoid using abbreviations unless the full name is excessive.
10. Avoid abbreviations longer than 5 characters.
11. Any Abbreviations must be widely known and accepted.
12. Use uppercase for two-letter abbreviations, and Pascal Case for longer abbreviations.
13. Do not use C# reserved words as names.
14. Avoid naming conflicts with existing .NET Framework namespaces, or types.
15. Do not include the parent class name within a property name.

Example: Customer.Name NOT Customer.CustomerName

1. Try to prefix Boolean variables and properties with “Can”, “Is” or “Has”.
2. Append computational qualifiers to variable names like Average, Count, Sum, Min, and Max where appropriate.
3. When defining a root namespace, use a Product, Company, or Developer Name as the root.

Example: LanceHunt.StringUtilities

## C# Formatting

1. Never declare more than 1 namespace per file.
2. Avoid putting multiple classes in a single file.
3. Always place curly braces ({ and }) on a new line.
4. Always use curly braces ({ and }) in conditional statements.
5. Always use a Tab & Indention size of 4.
6. Declare each variable independently – not in the same statement.
7. Place namespace “using” statements together at the top of file. Group .NET namespaces above custom namespaces.
8. Group internal class implementation by type in the following order:
   1. Member variables.
   2. Constructors & Finalizers.
   3. Nested Enums, Structs, and Classes.
   4. Properties
   5. Methods
9. Sequence declarations within type groups based upon access modifier and visibility:
   1. Public
   2. Protected
   3. Internal
   4. Private
10. Segregate interface Implementation by using #region statements.
11. Use white space (CR/LF, Tabs, etc) liberally to separate and organize code.
12. Place Assembly scope attribute declarations on a separate line.
13. Place Type scope attribute declarations on a separate line.
14. Place Method scope attribute declarations on a separate line.
15. Place Member scope attribute declarations on a separate line.
16. Place Parameter attribute declarations inline with the parameter.

# Client Application

## JavaScript

##### Variables

* 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
* Don't start names with a $ sign. It will put you in conflict with many JavaScript library names.

##### Avoid Global Variables

* Minimize the use of global variables.
* This includes all data types, objects, and functions.
* Global variables and functions can be overwritten by other scripts.
* Use local variables instead.

##### Initialize Variables

It is a good coding practice to initialize variables when you declare them.

This will:

* Give cleaner code
* Provide a single place to initialize variables
* Avoid undefined values
* Initializing variables provides an idea of the intended use (and intended data type).

Example:

// Declare and initiate at the beginning  
var firstName = "",  
    lastName = "",  
    price = 0,  
    discount = 0,  
    fullPrice = 0,  
    myArray = [],  
    myObject = {};

##### Never Declare Number, String, or Boolean Objects

Declaring these types as objects, slows down execution speed, and produces nasty side effects

Yes: var x = "John";

No: var y = new String("John");

* Use {} instead of new Object()
* Use "" instead of new String()
* Use 0 instead of new Number()
* Use false instead of new Boolean()
* Use [] instead of new Array()
* Use /()/ instead of new RegExp()
* Use function (){} instead of new function()

##### Brackets

* Open-brace ("{") characters on the same line as the statement that opens the block.
* Use one space before the opening bracket
* Put the closing bracket on a new line, without leading spaces

if (condition) {

/\* handle the condition \*/

} else {

/\* handle the "else" case \*/

}

##### Code Indentation

Always use 4 spaces for indentation of code blocks:

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

##### Line Length < 80 & Line breaks

* Long lines shouldn't be allowed - break at natural breaking points.
* For readability, avoid lines longer than 80 characters.
* If a JavaScript statement does not fit on one line, the best place to break it, is after an operator or a comma

Example 1:

if (class.CONDITION || class.OTHER\_CONDITION || class.SOME\_OTHER\_CONDITION

|| class.YET\_ANOTHER\_CONDITION ) {

/\* something \*/

}

var toolkitProfileService = Components.classes["@mozilla.org/toolkit/profile-service;1"]

.createInstance(Components.interfaces.nsIToolkitProfileService);

Example 2:

document.getElementById("demo").innerHTML =

"Hello Dolly.";

##### Object Rules

* 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.

Example:

var person = {  
    firstName: "John",  
    lastName: "Doe",  
    age: 50,  
    eyeColor: "blue"  
};

## AngularJS

#### Define 1 component per file

##### // app.module.js

angular

.module('app', ['ngRoute']);

##### // some.controller.js

angular

.module('app')

.controller('SomeController', SomeController);

function SomeController() { }

##### // someFactory.js

angular

.module('app')

.factory('someFactory', someFactory);

function someFactory() { }

#### Wrap Angular components in an Immediately Invoked Function Expression (IIFE)

##### // logger.js

(function() {

'use strict';

angular

.module('app')

.factory('logger', logger);

function logger() { }

})();

##### // storage.js

(function() {

'use strict';

angular

.module('app')

.factory('storage', storage);

function storage() { }

})();

### Module

#### Modules - Avoid Naming Collisions

Unique names help avoid module name collisions. Separators help define modules and their submodule hierarchy. For example app may be your root module while app.dashboard and app.users may be modules that are used as dependencies of app.

#### Declare modules without a variable using the setter syntax

No:

var app = angular.module('app', [

Yes:

angular

.module('app', [

'ngAnimate',

'ngRoute',

'app.shared',

'app.dashboard'

]);

#### When using a module, avoid using a variable and instead use chaining with the getter syntax

No:

var app = angular.module('app');

app.controller('SomeController', SomeController);

function SomeController() { }

Yes:

angular

.module('app')

.controller('SomeController', SomeController);

function SomeController() { }

#### Module - Only set once and get for all other instances

/\* recommended \*/

// to set a module

angular.module('app', []);

// to get a module

angular.module('app');

#### Use named functions instead of passing an anonymous function in as a callback

No:

/\* avoid \*/

angular

.module('app')

.controller('DashboardController', function() { })

.factory('logger', function() { });

Yes:  
// dashboard.js

angular

.module('app')

.controller('DashboardController', DashboardController);

function DashboardController() { }

// logger.js

angular

.module('app')

.factory('logger', logger);

function logger() { }

### Controllers

#### controllerAs View Syntax

Use the controllerAs syntax over the classic controller with $scope syntax

No:

<div ng-controller="CustomerController">

{{ name }}

</div>

function CustomerController($scope) {

$scope.name = {};

$scope.sendMessage = function() { };

}

Yes:

<div ng-controller="CustomerController as customer">

{{ customer.name }}

</div>

function CustomerController() {

this.name = {};

this.sendMessage = function() { };

}

#### Bindable Members Up Top

No:

function SessionsController() {

var vm = this;

vm.gotoSession = function() {

/\* ... \*/

};

vm.refresh = function() {

/\* ... \*/

};

vm.search = function() {

/\* ... \*/

};

vm.sessions = [];

vm.title = 'Sessions';

}

Yes:

function SessionsController() {

var vm = this;

vm.gotoSession = gotoSession;

vm.refresh = refresh;

vm.search = search;

vm.sessions = [];

vm.title = 'Sessions';

////////////

function gotoSession() {

/\* \*/

}

function refresh() {

/\* \*/

}

function search() {

/\* \*/

}

}



#### Defer Controller Logic to Services

Keeps the controller slim, trim, and focused

#### Keep Controllers Focused

Define a controller for a view, and try not to reuse the controller for other views. Instead, move reusable logic to factories and keep the controller simple and focused on its view.

#### Assigning Controllers

// route-config.js

angular

.module('app')

.config(config);

function config($routeProvider) {

$routeProvider

.when('/avengers', {

templateUrl: 'avengers.html',

controller: 'Avengers',

controllerAs: 'vm'

});

}

### Services

#### Accessible Members Up Top

/\* recommended \*/

function dataService() {

var someValue = '';

var service = {

save: save,

someValue: someValue,

validate: validate

};

return service;

////////////

function save() {

/\* \*/

};

function validate() {

/\* \*/

};

}

#### Function Declarations to Hide Implementation Details

function dataservice($http, $location, $q, exception, logger) {

var isPrimed = false;

var primePromise;

var service = {

getAvengersCast: getAvengersCast,

getAvengerCount: getAvengerCount,

getAvengers: getAvengers,

ready: ready

};

return service;

////////////

function getAvengers() {

// implementation details go here

}

…

function ready(nextPromises) {

// implementation details go here

}

}

### Directives

* Create one directive per file. Name the file for the directive.

Yes:

angular

.module('sales.order')

.directive('acmeOrderCalendarRange', orderCalendarRange);

function orderCalendarRange() {

/\* implementation details \*/

}

* When manipulating the DOM directly, use a directive.
* If the directive simply hides and shows, use ngHide/ngShow
* DOM manipulation can be difficult to test, debug, and there are often better ways (e.g. CSS, animations, templates)

## Typescript

## HTML

#### HTML elements

* **Correct**: the [<span>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/span" \o "The HTML <span> element is a generic inline container for phrasing content, which does not inherently represent anything. It can be used to group elements for styling purposes (using the class or id attributes), or because they share attribute values, such as ) element
* **Incorrect**: the span tag

#### Comments on closing tags

After every major thing of HTML, for example, the end of a carousel, or the end of the content <div>, place a closing-comment.

Example:

...

</div><!-- /carousel -->

...

</div><!-- /content -->

#### Namespaced fragment identifiers

Nice way to add a little more meaning to your fragment identifiers and give a little bit more of a clue as to what they actually link to.

Example:

<a href=#section:fragment-identifiers>Fragment identifiers</a>

...

<div id=section:fragment-identifiers>...</div>

* Use Correct Document Type: <!DOCTYPE html>
* Use Lower Case Element Names: <section></section>
* Close All HTML Elements: <p>This is a paragraph.</p>
* Close Empty HTML Elements: <meta charset="utf-8" />

#### Image Attributes

* Always use the alt attribute with images. It is important when the image cannot be viewed.
* Always define image size. It reduces flickering because the browser can reserve space for images before they are loaded.

<img src="html5.gif" **alt="HTML5"** **style="width:128px;height:128px"**>

#### Title and Meta Data

* The <title> element is required in HTML5. Make the title as meaningful as possible.
* **To ensure proper interpretation, and correct search engine indexing, both the language and the character encoding should be defined**

<!DOCTYPE html>

<html lang="en-US">

<head>

**<meta charset="UTF-8">**

**<title>HTML5 Syntax and Coding Style</title>**

</head>

## CSS

* Place the opening bracket on the same line as the selector.
* Use one space before the opening bracket.
* Use 2 spaces of indentation.
* Use colon plus one space between each property and its value.
* Use space after each comma or semicolon.
* Use semicolon after each property-value pair, including the last.
* Only use quotes around values if the value contains spaces.
* Place the closing bracket on a new line, without leading spaces.
* Avoid lines over 80 characters.

#### Declaration order

##### /\* Positioning \*/

position: absolute;

z-index: 10;

top: 0;

right: 0;

bottom: 0;

left: 0;

##### /\* Display & Box Model \*/

display: inline-block;

overflow: hidden;

box-sizing: border-box;

width: 100px;

height: 100px;

padding: 10px;

border: 10px solid #333;

margin: 10px;

vertical-align: top;

white-space: nowrap;

##### /\* Other \*/

background: #000;

color: #fff;

font-family: sans-serif;

font-size: 16px;

text-align: right;

#### No IDs

This is more a technical thing, but I have a blanket-ban on IDs in CSS. There is literally no point in them, and they only ever cause harm. Everything that needs styling is done so without using IDs.

#### Table of contents

At the top of my CSS files make table of contents that maps to the section titles in the document:

/\*------------------------------------\*\

CONTENTS

\\*------------------------------------\*/

/\*

NOTES

RESET

SHARED Share anything we can across elements.

MAIN HTML, BODY, etc.

\*/

#### Section titles

Denote each section (layout, type, tables etc) in CSS:

/\*------------------------------------\*\

$MAIN

\\*------------------------------------\*/

/\*------------------------------------\*\

$NAVIGATION

\\*------------------------------------\*/

This section heading is prepended with a $. When try to find for a section – find for $MAIN and not MAIN. A search for $MAIN will only ever find a section.

##### Title and heading capitalization

Use sentence-style capitalization (only capitalize the first word and proper nouns):

* **Correct**: "A new method for creating JavaScript rollovers"
* **Incorrect**: "A New Method for Creating JavaScript Rollovers"

### Latin abbreviations

#### In notes and parentheses

Common Latin abbreviations (etc., i.e., e.g.) may be used in parenthetical expressions and notes. Use periods in these abbreviations.

* Correct: Web browsers (e.g. Firefox) can be used ...
* Incorrect: Web browsers e.g. Firefox can be used ...
* Incorrect: Web browsers, e.g. Firefox, can be used ...
* Incorrect: Web browsers, (eg: Firefox) can be used ...

#### In running text

In regular text (i.e. text outside of notes or parentheses), use the English equivalent of the abbreviation.

* + **Correct**: ... web browsers, and so on.
  + **Incorrect**: ... web browsers, etc.
  + **Correct**: Web browsers such as Firefox can be used ...
  + **Incorrect**: Web browsers e.g. Firefox can be used ...

#### Meanings and English equivalents of Latin abbreviations

|  |  |  |
| --- | --- | --- |
| **Abbrev** | **Latin** | **English** |
| cf. | confer | compare |
| e.g. | exempli gratia | for example |
| et al. | et alii | and others |
| etc. | et cetera | and so forth, and so on |
| i.e. | id est | that is, in other words |
| N.B. | nota bene | note well |
| P.S. | post scriptum | postscript |

#### Plurals of acronyms and abbreviations

For plurals of acronyms or abbreviations, add s. Don't use an apostrophe. Ever. Please.

* **Correct**: CD-ROMs
* **Incorrect**: CD-ROM's

#### Hyphenation

Hyphenate compounds when the last letter of the prefix is a vowel and is the same as the first letter of the root.

* **Correct**: email, re-elect, co-op
* **Incorrect**: e-mail, reelect, coop

#### Dates

For dates (not including dates in code samples) use the format "January 1, 1990":

* **Correct**: February 24, 2006
* **Incorrect**: February 24th, 2006; 24 February, 2006; 24/02/2006

Alternately, you can use the YYYY/MM/DD format:

* **Correct**: 2006/02/24
* **Incorrect**: 02/24/2006; 24/02/2006; 02/24/06

#### Decades

For decades, use the format "1990s". Don't use an apostrophe.

* **Correct**: 1990s
* **Incorrect**: 1990's

#### Plurals of numerals

For plurals of numerals add "s". Don't use an apostrophe.

* **Correct**: 486s
* **Incorrect**: 486's

#### Commas

In running text, use commas only in five-digit and larger numbers.

* **Correct**: 4000; 54,000
* **Incorrect**: 4,000; 54000

#### Serial comma

**Use the serial comma**. The serial (also known as "Oxford") comma is the comma that appears before the conjunction in a series of three or more items.

* **Correct**: I will travel on trains, planes, and automobiles.
* **Incorrect**: I will travel on trains, planes and automobiles.

## Structure

ASP.NET Structure <- AngularJS & Typescript

ASP.NET Structure

AngularJS

Typescript

* Solution *Name*
* ***Name*Application**
* Properties
* References
* App\_Data
* Config
* Content
* Fonts
  + - Images
    - Scripts
      * Angular*Name*Project
* app.module.ts
* app.routes.ts
* Controllers
* ts*Name*Controller.ts

…

* Interfaces
* interfaces.ts

…

* Services
  + *name*Services.ts

…

* Views
  + partial.html

…

* + - * includeScript
* typings
* angularjs
* jquery

…

* Styles
  + less
* style.less

…

* + css
* style.min.css

…

* + includeStyles
* Controller
* *ProjectName*Controller.cs
* Services
* Models
* Views
  + - ProjectName
  + Index.cshtml
* Shared
  + \_Layout.cshtml
* \_ViewStart.cshtml
* web.config
* Global.asax
* packages.config.
* Web.config

# Explanation

Reasons for using a naming convention

* to reduce the effort needed to read and understand source code;[[1]](https://en.wikipedia.org/wiki/Naming_convention_(programming)#cite_note-1)
* to enable code reviews to focus on more important issues than arguing over syntax and naming standards.
* to enable code quality review tools to focus their reporting mainly on significant issues other than syntax and style preferences.
* to enhance source code appearance (for example, by disallowing overlong names or unclear abbreviations).

Documentation allows you to transfer the why behind code. Much in the same way code comments explain the why, and not the how, documentation serves the same purpose.

If people don’t know why your project exists,

they won’t use it.

If people can’t figure out how to install your code,

they won’t use it.

If people can’t figure out how to use your code,

they won’t use it.

You only get contributions after you have put in a lot of work.

You only get contributions after you have users.

You only get contributions after you have documentation.

Technical writing is an art that doesn’t come naturally. Writing documentation will start you down the road to being a better technical writer, which is a useful skill to have as a programmer.

### Documentation of Project:

What problem your project solves

Installation instructions

# References

*Code Magazin*. (n.d.). Retrieved from http://www.codemag.com/article/1405071

*CSS Wizard*. (n.d.). Retrieved from http://csswizardry.com/2012/04/my-html-css-coding-style/

*Dofactory*. (n.d.). Retrieved from http://www.dofactory.com/reference/csharp-coding-standards

*DZone*. (n.d.). Retrieved from https://dzone.com/articles/best-practices-variable-and

*Google Github*. (n.d.). Retrieved from https://google.github.io/styleguide/javascriptguide.xml

Hunt, L. (n.d.). *C# Coding Standards for .NET*. Retrieved from http://se.inf.ethz.ch/old/teaching/ss2007/251-0290-00/project/CSharpCodingStandards.pdf

*Idiomatic Css*. (n.d.). Retrieved from https://github.com/necolas/idiomatic-css

*Javarevisited*. (n.d.). Retrieved from http://javarevisited.blogspot.rs/2014/10/10-java-best-practices-to-name-variables-methods-classes-packages.html

*Jetbrains*. (n.d.). Retrieved from https://www.jetbrains.com/webstorm/help/code-style-typescript.html

*John Papa Style Guide*. (n.d.). Retrieved from https://github.com/johnpapa/angular-styleguide

*JQuery Fondation - JavaScript Style Guide*. (n.d.). Retrieved from https://contribute.jquery.org/style-guide/js/

*MSDN C# code style guide*. (n.d.). Retrieved from https://msdn.microsoft.com/en-us/library/ff926074.aspx

*W3School*. (n.d.). Retrieved from http://www.w3schools.com/js/js\_conventions.asp

*W3School HTML5*. (n.d.). Retrieved from http://www.w3schools.com/html/html5\_syntax.asp

*Wikipedia*. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Naming\_convention\_(programming)