Matter Center for Office 365

Coding guidelines

Table of contents

[C# Naming Guidelines 3](#_Toc438039584)

[C# Code Commenting 4](#_Toc438039585)

[C# Flow Control: 6](#_Toc438039586)

[General Guidelines: 7](#_Toc438039587)

[Anti-XSS (Cross site scripting vulnerabilities) 8](#_Toc438039588)

[Exception Handling: 9](#_Toc438039589)

[Essential Tools: 10](#_Toc438039590)

[HTML Guidelines 11](#_Toc438039591)

[CSS Guidelines: 11](#_Toc438039592)

[JavaScript coding Guidelines 13](#_Toc438039593)

[Naming convention: 13](#_Toc438039594)

[General Guidelines 14](#_Toc438039595)

[JQuery Guidelines 16](#_Toc438039596)

# C# Naming Guidelines

|  |
| --- |
| “c” = camel Case  “P” = Pascal Case  “\_” = Prefix with \_Underscore  “x” = Not Applicable |

| **Identifier** | **Public** | **Protected** | **Internal** | **Private** | **Notes** |
| --- | --- | --- | --- | --- | --- |
| Method | P | P | P | P | Use a Verb or Verb-Object pair. |
| Property | P | P | P | P | Do not prefix with Get or Set. |
| Field | P | P | P | c | Only use Private fields.  No Hungarian Notation! |
| Constant | x | x | x | x | Constants should be all in Uppercase with words separated by underscore.  Example:  public class SomeClass  {      const int DEFAULT\_SIZE = 100;  } |
| Static Field | P | P | P |  |  |
| Enum | P | P | P | P | Options are also Pascal Case |
| Delegate | P | P | P | P |  |
| Event | P | P | P | P |  |
| Parameter | x | x | x | c |  |
| Project File | P | x | x | x | Match Assembly & Namespace. |
| Source File | P | x | x | x | Match contained class. |
| Class | P | P | P | P |  |
| Structure | P | P | P | P |  |
| Interface | P | P | P | P | Prefix with a capital I. |
| Generic Class | P | P | P | P | Use T or K as Type identifier    //correct  public class LinkedList<K,T>  {...}    //incorrect  public class LinkedList<KeyType,DataType>  {...} |
| Attribute Class | P | P | P | P | Suffix custom attribute classes with Attribute |
| Exception Class | P | P | P | P | Suffix custom exception classes with Exception |
| Resources | P | x | x | x | Use Pascal casing in resource keys |
| Local Variable within Methods | c | c | c | c |  |
| Global Variable | P |  |  |  |  |

# C# Code Commenting

1. Run StyleCop to identify the commenting related issues
2. Indent comments at the same level of indentation as the code you are documenting
3. Run spell check on all comments. Misspelled comments indicate sloppy development. Best way to cover for this is the spell checker add-in available in Visual Studio 2010 and higher.
4. Write all comments in the same language, be grammatically correct, and use appropriate punctuation
5. Use // or /// but never /\* … \*/
6. Use inline-comments to explain assumptions, known issues, and algorithm insights
7. Do not use inline-comments to explain obvious code. Well written code is self-documenting
8. Only use comments for bad code to say “fix this code” – otherwise remove, or rewrite the code
9. Include comments using Task-List keyword flags to allow comment-filtering

**Example:**

|  |
| --- |
| // TODO: Place Database Code Here  // UNDONE: Removed P\Invoke Call due to errors  // HACK: Temporary fix until able to refactor |

1. Always apply C# comment-blocks (///) to public, protected, and internal declarations.
2. Only use C# comment-blocks for documenting the API
3. Always include <summary> comments. Include <param>, <return>, and <exception> comment sections where applicable
4. Use Section tags to define different sections within the type documentation.

| **Section Tags** | **Description** | **Location** |
| --- | --- | --- |
| <summary> | Short description | type or member |
| <remarks> | Describes preconditions and other additional information. | type or member |
| <param> | Describes the parameters of a method | Method |
| <returns> | Describes the return value of a method | Method |
| <exception> | Lists the exceptions that a method or property can throw | method, even or  property |
| <value> | Describes the type of the data a property accepts and/or returns | Property |
| <example> | Contains examples (code or text) related to a member or a type | type or member |
| <seealso> | Adds an entry to the See Also section | type or member |
| <overloads> | Provides a summary for multiple overloads of a method | First method in an overload list. |

1. Provide the following information at the top of each page. This information is mandatory:

|  |  |
| --- | --- |
| **File** | Name of the file with extension. |
| **Project:** | Name of the project. |
| **Solution:** | Name of the solution. |
| **Author:** | Author who created this page. |
| **Date:** | Date of page creation. |
| **Description:** | Brief summary of the page. |
| **Change History:** | This provides the information about changes made to the file after creation. |

Typical example is given below.

|  |
| --- |
| #region Page Summary  /// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  ///  /// File:     BasePage.cs  /// Project:     ConMan  /// Solution:     ConMan  ///  /// Author:    \_\_\_\_\_\_  /// Date:    December 22, 2006  /// Description: This file defines the BasePage for all other pages.  /// Contains functionality common to all the pages.  ///  /// Change History:  /// Name        Date            Version        Description  /// -------------------------------------------------------------------------------  /// \_\_\_\_\_\_\_\_\_\_\_\_    March 05, 2005        1.0.1.2        Created    /// -------------------------------------------------------------------------------  /// Copyright (C) <copyright information of the client>.  /// -------------------------------------------------------------------------------  #endregion |

# C# Flow Control:

1. Use the **ternary** conditional operator only for trivial conditions. Avoid complex or compound ternary operations.

**Example**:

|  |
| --- |
| int result = isValid ? 9 : 4; |

1. Avoid evaluating Boolean conditions against true or false.

**Example**:

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| if (isValid)  {…} | if (isValid == true)  {…} |

1. Avoid assignment within conditional statements

**Example:**

|  |
| --- |
| if ((i=2)==2) {…} |

1. Avoid compound conditional expressions – use Boolean variables to split parts into multiple manageable expressions.

**Example:**

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| isHighScore = (value >= \_highScore);  isTiedHigh = (value == \_highScore);  isValid = (value < \_maxValue);  if ((isHighScore && ! isTiedHigh) && isValid)  {…} | if (((value > \_highScore) && (value != \_highScore)) && (value < \_maxScore))  {…} |

1. Only use switch/case statements for simple operations with parallel conditional logic.
2. Prefer nested if/else over switch/case for short conditional sequences and complex conditions.
3. Prefer polymorphism over switch/case to encapsulate and delegate complex operations. Use TryParse for conversions like integer, decimal, date etc. in place of direct conversions like ToInt32(), ToDatetime() etc.

# General Guidelines:

1. Do not omit access modifiers. Explicitly declare all identifiers with the appropriate access modifier instead of allowing the default.

**Example:**

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| private Void WriteEvent(string message)  {…} | Void WriteEvent(string message)  {…} |

1. Avoid explicit properties that do nothing but access a member variable. Use automatic properties instead.

|  |  |
| --- | --- |
| **INCORRECT** | **CORRECT** |
| Class MyClass  {  int m\_Number;  Public int Number  {  get{Return number;}  set{number = value;}  }  } | Class MyClass  {  public int Number  {get; set;}  } |

1. Never hardcode strings that will be presented to end users. Use resources instead
2. Use StringBuilder class instead of String when you have to manipulate string objects in a loop.
3. Run StyleCop to check formatting
4. Never declare more than one namespace per file
5. Avoid putting multiple classes in a single file
6. Always place curly braces ({ and }) on a new line
7. Always use curly braces ({ and }) in conditional statements
8. Place namespace “using” statements together at the top of file. Group .NET namespaces above custom namespaces
9. Use #region to group related pieces of code together.
10. Only declare related attribute declarations on a single line, otherwise stack each attribute as a separate declaration.

**Example:**

|  |  |
| --- | --- |
| **CORRECT** | **INCORRECT** |
| [Attrbute1, RelatedAttribute2]  [Attrbute3]  [Attrbute4]  public class MyClass  {…} | [Attrbute1, Attrbute2, Attrbute3]  public class MyClass  {…} |

1. Always prefer C# Generic collection types over standard or strong-typed collections.
2. Prefer String.Format() or StringBuilder over string concatenation.
3. Do not compare strings to String.Empty or “” to check for empty strings. Instead, compare by using String.Length == 0
4. Avoid explicit casting. Use the as operator to defensively cast to a type.

|  |
| --- |
| Dog dog = new GermanShepherd();  GermanShepherd shepherd = dog as GermanShepherd;  if(shepherd != null)  {...} |

1. Constant variables do not allocate memory. So it is always recommended to use constant variables instead of static variables
2. **For example**:  const int NUMBER\_SAMPLE= 4;
3. Read-only static variables can be assigned a dynamic value during initialization which is not possible with constants
4. E.g. private readonly static int TotalCount = //Some method which returns the count
5. Also, constants are not allocated memory. When an application is compiled, the value of the constant is embedded in the assembly during compile time. Now if the dll containing the Constant changes the value of the constant, the main application has to be recompiled to reflect the new constant value. However, in case of read only static variables, their value is looked up at runtime and hence no recompilation of the main application is required.

# Anti-XSS (Cross site scripting vulnerabilities)

1. To avoid cross-site scripting there are two things that need to be done:

* Verify Input - Constrain the acceptable range of input characters.
* Encode Input - Use HttpUtlitity.HtmlEncode when displaying input back to user.

1. Encode Input using HttpUtility.HtmlEncode
2. Use the HttpUtility.HtmlEncode method to encode output if it contains input from the user, such as input from form fields, query strings, and cookies or from other sources, such as databases. Do not write the response back without validating or encoding the data.

**Example:**

|  |
| --- |
| Response.Write(HttpUtility.HtmlEncode(Request.Form["name"])); |

1. Use AntiXSS base class library for ASP.NET 4.5 projects.

**Example:**

|  |
| --- |
| AntiXssEncoder.HtmlEncode("dasd", true); |

# Exception Handling:

1. Do not use try/catch blocks for flow-control
2. catch only those exceptions that you can handle
3. Never declare an empty catch block
4. Avoid nesting a try/catch within a catch block
5. While re-throwing an exception, preserve the original call stack by omitting the exception argument from the throw statement.

**Example:**

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| catch(Exception)  {  Log(ex);  throw;  } | catch(Exception ex)  {  Log(ex);  throw ex;  } |

1. Use the finally block to release resources from a try statement.
2. Always use validation to avoid exceptions.

**Example:**

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| if(conn.State != ConnectionState.Closed)  {  conn.Close();  } | try  {  conn.Close();  }  Catch(Exception ex)  {  // handle exception if already closed!  } |

1. Always set the innerException property on thrown exceptions so the exception chain & call stack are maintained.
2. Avoid defining custom exception classes. Use existing exception classes instead.

# Essential Tools:

|  |  |  |  |
| --- | --- | --- | --- |
| Visual Studio 2013  Code Analyzer | 2013  (Update 2) | Machine generated alternative text: Application  Build  Web  Package/PubIish Web  Package/PubIish SQL  Silverlight Applications  Build Events  Resources  Settings  Reference Paths  Si nin  Code Analysis*  Code Contracts  TypeScript Build  Configuration: Active (Debug)  V Enable Code Analysis on Build  Platform:  Active (Any CPU)  Open  Suppress results from generated code (managed only)  Rule Set  Microsoft All Rules  Microsoft All Rules  Microsoft Basic Design Guideline Rules  Microsoft Extended Correctness Rules  Microsoft Extended Design Guideline Rules  Microsoft Globalization Rules  Microsoft Managed Minimum Rules  Microsoft Managed Recommended Rules  Microsoft Mixed (C++  /CLR) Minimum Rules  Microsoft Mixed (C++  /CLR) Recommended Rules  Microsoft Security Rules  Growse..»  «Choose multi le rule sets..» | NA |
| StyleCop | 4.7 | Analyzes C# source code to enforce a set of style and consistency rules | [Web](http://stylecop.codeplex.com/releases/view/79972) |
| Visual Studio 2013 Code Clone | 2013  (Update 2) | Machine generated alternative text: ANALYZE WINDOW HELP  Performance and Diagnostics  Launch Node.js Profiling..  Profiler  Run Code Analysis on Solution  Configure Code Analysis for Solution  Calculate Code Metrics for Selected Project(s)  Analyze Solution for Code Clones  In CWS  Alt*F2  Alt*F11 | NA |

# HTML Guidelines

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Use of  HTML5 Block Level Elements | Instead of staying with divs for everything, take some time to learn the new HTML 5 elements like <header>,<footer>, <article> and others. They work the same way but improve readability   |  |  | | --- | --- | | /\* Not recommended \*/ | /\* Recommended \*/ | | <body>     <div id="header">         ...     </div>     <div id="main">             ...     </div>     <div id="footer">         ...     </div> </body> | <body>     <header>         ...     </header>     <article>         ...         <section>             ...         </section>         ...     </article>     <footer>         ...     </footer> </body> | |
| Client side data storage | Use local storage or session storage introduced with HTML5 rather than cookies  Reference: <http://technobytz.com/cookies-vs-html-5-web-storage-comparison.html>  Always check applicability of storage type for requirement in hand |
| ALT attribute for image tag | Always use ALT attribute with an image tag. This will help in case image does not load because of connection issues |

# CSS Guidelines:

| **Title** | **Description** |
| --- | --- |
| Avoid use of inline CSS | Never use inline CSS, always separate your content from presentation |
| Placement of CSS file | Place your CSS file reference in a head tag of HTML page  or  If Header is not present, include reference before other page content (In cases like SharePoint content editor web part) |
| Property ordering | Put declarations in alphabetical order in order to achieve consistent code in a way that is easy to remember and maintain.  background: fuchsia; border: 1px solid; color: black; text-align: center;    One more suggestion is to group it logically in following order   * + Display   + Positioning   + Box model   + Colors and Typography   + Other   You can select any of these, but make sure that it is consistent. |
| Naming convention | Use Pascal naming convention.  Avoid using class names like “purple”, “big”, “largeText” or “margin50”. Class names should describe the element or collection, not the style - the CSS properties describe the style. If that color changes you would have to modify your HTML to change the appearance, which defeats the idea of separation of styles from content. Or worse, you would have a class name called “purple” when the background-color might be declared red. |
| Organizing style sheet | Organize the Stylesheet with a Top-down Structure. It always makes sense to lay your CSS file out in a way that allows you to quickly find parts of your code. One option is a top-down format that tackles styles as they appear in the source code. So, an example CSS file might be ordered like this:  Generic classes (body, a, p, h1, etc.)  #header  #nav-menu  #main-content |
| Shorthand Properties | CSS offers a variety of 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 \*/ | /\* 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; | border-top: 0; font: 100%/1.6 palatino, georgia, serif; padding: 0 1em 2em; | |
| Avoid a universal key selector | Never apply styles using universal selector   |  | | --- | | /\* Not recommended \*/ | | \* {   margin: 0;   padding: 0; } |   Use all CSS relative to topmost element created for report |
| Remove redundant qualifiers | These qualifiers are redundant:   * 1. ID selectors qualified by class and/or tag selectors   2. Class selectors qualified by tag selectors (when a class is only used for one tag, which is a good design practice anyway)   3. It's easy to unknowingly add extra selectors to our CSS that clutters the stylesheet.   In some cases you need to add this to apply more specific styles, but in most of the case there will be optimal way to achieve this   |  |  | | --- | --- | | /\* Not recommended \*/ | /\* Recommended \*/ | | ul#example {} div.error {} | #example {} .error {} | | body #container .someclass ul li {....} | .someclass li {...} | |

# JavaScript coding Guidelines

## Naming convention:

1. Names should not be too small like ex1, i, j, k etc. or should not be very large. Iterator variables in the loops can be an exception here.
2. Name should be simple, easy, readable and most importantly SHORT
3. Names should not start with a numeral (0-9). Such names will be illegal.
4. **Variable Names:**
5. Use Lower Camel Case (variableNameShouldBeLikeThis) for variable names.

* Name should start with single letter which defines it's data type. This helps the developer to know the behavior of the variable and gets an idea about what kind of data it has in it.
* Refer to the table below for some of the data types widely used along with the conventions.

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Data Type** | **Prefix letter** | **Sample Variable Name** |
| 1 | int | i | iCounter |
| 2 | Float | f | fPrice |
| 3 | Boolean | b | bFlag |
| 4 | String | s | sFullName |
| 5 | Object | o | oDiv |

1. Constants: There is nothing as constants in JavaScript. But if you want for particular variable developer should not try to modify the value then the name should be in all capital letters (CONSTANTNAME). Specify the prefix similar to above table.
2. **Function Names / Method Names:**
3. JavaScript function/method names should use Lower Camel Casing (functionNamesLikeThis)
   * In case your function returns a value. Prefix the function name with the letter indicating the type of values returned. Refer table mentioned above for Prefixing a letter for variable names.
   * Function names should be intuitive and should use concise words explaining the existence of the function.
   * **For e.g.:** if you want to name a code block that checks whether the user is of legal age or not for driving, you name the function as  “isAboveEighteen()”  while this seems to serve the purpose but a more appropriate function name would be “isLegalAge()”.

## General Guidelines

1. Declarations with var: Always
2. Prefer ' over "
3. JS allows both single quotes or double quotes
4. But for consistency single-quotes (') are preferred to double-quotes ("). This is helpful when creating strings that include HTML.

var msg = 'This is some "HTML"';

1. Always use semicolons to terminate JS statements
2. Cache the length in the loops.

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| var foo = document.getElementsByTagName('p');  for(var i=0, len=foo.length; i<len; i++) {}; | var foo = document.getElementsByTagName('p');  for(var i=0; i<foo.length; i++) {}; |

1. Use {} Instead of new Object()

**Example:**

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| var o = {     name: 'Jeffrey',     lastName = 'Way',     someFunction : function() {        console.log(this.name);     }}; | var o = new Object();  o.name = 'Jeffrey';  o.lastName = 'Way';  o.someFunction = function() {     console.log(this.name);  } |

1. Use [] Instead of new Array()
2. Always use strict comparison ( === Instead of == )
3. Using try/catch (Note: Exceptions Are For Exceptional Cases)

* Try/catch are expensive
* Do not use nested try/catch, instead use try/catch at the topmost level
* Do not ignore exceptions

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| try {      doStuff();  } catch(ignore) {      log(ignore);  } | try {      doStuff();  } catch(ignore) {      // Do nothing, just ignore.  } |

* Do not use try/catch within loops.

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| try {      while(condition) {          stuff();      }  } catch(e) {      log(e);  } | while(condition) {      try {          stuff();      } catch(e) {          log(e);      }  } |

1. Curly Love, use curly braces it even if it is not necessary.

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| var bCheck = true;  if(bCheck) {  alert(bCheck);  }  else {  alert(bCheck);  } | var bCheck = true;  if(bCheck)  alert(bCheck);  else  alert(bCheck); |

1. Minimize DOM access: Accessing DOM elements with JavaScript is slow so in order to have a more responsive page, you should:

Cache references to accessed elements:

var msg = document.getElementById('someDiv') //store(cache) the object in a variable

if (msg)

{

msg.style.display = 'none'

}

1. Perform null checks for the object before accessing or updating any of its properties. Refer below the sample for same:

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| var msg = document.getElementById('someWrongDivName')  if (msg)  {  msg.style.display = 'none'  }  else  {  alert('No error shown')  } | var msg = document.getElementById('someWrongDivName')    msg.style.display = 'none' //msg is null here |

# JQuery Guidelines

1. Use ID selector whenever possible. Finding a DOM element by its ID is the fastest way, both in JavaScript and in jQuery. Whenever possible, you should always use the ID selector instead of using classes or tag names, or other ways.
2. Avoid Loops. Nested DOM Selectors can perform better. Avoid unnecessary loops. If possible, use the selector engine to address the elements that are needed
3. Don't mix CSS with jQuery
4. Avoid multiple $(document).ready() calls
5. $.ajax performs a massive amount of work to allow us the ability to successfully make asynchronous requests across all browsers. You can use $.ajax method directly and exclusively for all your AJAX requests
6. If we have an element with id “refiner” and we want to add two classes “addColor” and “addBackground”, we can do it by putting the two class names in addClass method separated by a space.
7. Leverage Event Delegation (a.k.a. Bubbling). Every event (e.g. click, mouseover, etc.) in JavaScript “bubbles” up the DOM tree to parent elements. This is incredibly useful when we want many elements (nodes) to call the same function. Instead of binding an event listener function too many nodes — very inefficient — you can bind it once to their parent, and have it figure out which node triggered the event. For example, say we are developing a large form with many inputs, and want to toggle a class name when selected.

|  |
| --- |
| **// Inefficient**  $('#myList li).bind('click', function(){  $(this).addClass('clicked');  // do stuff  });    **// Instead, we should listen for the click event at the parent level:**  $('#myList).bind('click', function(e){  var target = e.target, // e.target grabs the node that triggered the event.  $target = $(target);  // wraps the node in a jQuery object  if (target.nodeName === 'LI') {  $target.addClass('clicked');  // do stuff  }  }); |