

|  |  |
| --- | --- |
| **Author** | Jeremy Caney (Jeremy@ignia.com) |
| **Version** | 01.00.00 |
| **Audience** | Ignia Development Staff |

# Overview

The purpose of this document is to define Ignia’s style guidelines for development practices when writing .NET code in C#. This document should make no assumptions of the application or environment, although generally it will be applied to web pages.

*It is expected that developers writing class libraries will be familiar with the Microsoft Class Library Design Guidelines, which should be considered a prerequisite to this document.*

## Revision History

|  |  |  |
| --- | --- | --- |
| **Author** | **Date** | **Comments** |
| Jeremy Caney | 02.03.05 | Initial version created |
| Jeremy Caney | 05.08.06 | Minor copy edits; no revisions to actual standards. |
| Jeremy Caney | 07.16.07 | Standardized portions with Microsoft’s internal standards.  Changed standards on Hungarian Notation. |

## Related Documents

1. Class Library Design Guidelines ([[Reference](ms-help://MS.NETFramework.v20.en/dv_fxdesignguide/html/418b3cb8-c26a-44f8-85da-acc5b8135edd.htm)], .NET SDK, Microsoft)

# Contents

[Contents 2](#_Toc172389689)

[Capitalization 3](#_Toc172389690)

[Summary 3](#_Toc172389691)

[Public Fields 3](#_Toc172389692)

[Client Ambiguity Exception 3](#_Toc172389693)

[Abbreviations 3](#_Toc172389694)

[Acronyms 3](#_Toc172389695)

[Case Sensitivity 4](#_Toc172389696)

[Naming Conventions 4](#_Toc172389697)

[Namespaces 4](#_Toc172389698)

[Methods 4](#_Toc172389699)

[Private Fields 4](#_Toc172389700)

[Hungarian Notation 5](#_Toc172389701)

[Form Fields 5](#_Toc172389702)

[Indentation and Formatting 5](#_Toc172389703)

[Indentation 5](#_Toc172389704)

[Spacing 5](#_Toc172389705)

[Bracing 6](#_Toc172389706)

[Example 6](#_Toc172389707)

[Comments 6](#_Toc172389708)

[Header Block 6](#_Toc172389709)

[Member Block 7](#_Toc172389710)

[Section Titles 7](#_Toc172389711)

[Inline Comments 7](#_Toc172389712)

[Editorial Comments 8](#_Toc172389713)

[Organization 8](#_Toc172389714)

[Conclusion 8](#_Toc172389715)

[Works Cited 8](#_Toc172389716)

# Capitalization

Ignia follows Microsoft’s guidelines for capitalization as outlined in the Class Library Design Guidelines as well as their Internal Coding Guidelines (Microsoft Corporation, 2005). At a high level, variables and parameters should use camelCase whereas everything else (namespaces, classes, members, etc) should be PascalCase. This section summarizes Microsoft’s standards for capitalization.

## Summary

|  |  |  |
| --- | --- | --- |
|  | *Capitalization Format* | *Example* |
| Namespace | PascalCase | Ignia.Web.Tools |
| Class Name | PascalCase | Utility |
| Members | PascalCase | Method(), Property |
| Member Variables | camelCase | \_email |
| Local Variables | camelCase | email |
| Parameter | camelCase | email |
| Interfaces | PascalCase | IInterface |
| Form Fields | *Varies; See “Naming Conventions”* | Country, btnSubmit |

## Public Fields

Microsoft recommends against public fields[[1]](#footnote-1); instead, fields should be exposed via properties. As a result, fields should always be private and thus camelCase; for information on naming conventions, see “Private Fields” below. The one exception to this is User Controls (including Master Pages) where, for simplicity, fields may be made public; since public fields act as properties, they should use PascalCase. Web forms (ASPX pages) may use either format, although generally all members implemented at the page level should be treated as private and thus fields should be camelCase.

## Client Ambiguity Exception

If the name of the field will be exposed to another development environment (such as an HTML Server Control) then it is permitted for the capitalization to follow the naming and capitalization style standards dictated by the client application (if applicable) to maintain consistency with client scripts; this is particularly an issue with web pages, where server controls (and their properties) may be exposed as XHTML and be accessible via JavaScript.

## Abbreviations

Abbreviations should not be used in identifiers. The two exceptions to this are “ID” and “OK”. When used as an identifier, these abbreviations should be capitalized (e.g., “ID”); however, when *contained* in an identifier they should use PascalCase (e.g., “UserId”). The exception to this is when used as the first (or only) word in a camelCase identifier, in which they should be lower case (e.g., “id”).

## Acronyms

Popular industry acronyms may be used as part of standard names. Only the first letter of an acronym should be capitalized, unless:

1. It is used as the first word in a camelCase identifier (in which it should be lower case) *or*
2. The acronym is only two letters (e.g., “IO”, “DB”) (in which it should be capitalized).

### Examples

|  |  |
| --- | --- |
| *PascalCase* | *camelCase (first word)* |
| Xml (e.g., XmlDocument) | xml (e.g., “xmlWriter”) |
| IO (e.g., “System.IO”) | io (e.g., “ioChannel”) |

## Case Sensitivity

Case sensitivity should not be relied on to distinguish elements; e.g., do not have two different methods named GetResources() and Getresources(). If capitalization rules are applied consistently then this should not be an issue.

# Naming Conventions

The following naming conventions should be adhered to in conjunction with the above capitalization rules. In additional to these conventions, developers working on class libraries should be familiar with Microsoft’s recommended naming conventions for various class identifiers (Microsoft Corporation, 2005).

## Namespaces

Namespaces should follow the convention CompanyName.TechnologyName[.Feature][.Design]. For instance, Ignia.Toolbox (Microsoft Corporation, 2005).

## Methods

Method names should typically consist of a verb (“add”, “delete”, “get”) as a means of differentiating them from properties (which should be nouns). In addition, Ignia commonly uses the following conventions:

|  |  |  |
| --- | --- | --- |
| Get | Retrieve a string | e.g., GetHeaderTag() |
| Write | Outputs a string | e.g., WriteHeaderTag() |

This can be valuable if you want multiple methods which output the same information in different formats (e.g., in a web application the Write will do a Response.Write() where as the Get will return a string). Example:

public String GetHeaderTag(String siteName) {

return (siteName);

}

public void WriteHeaderTag(String siteName) {

Response.Write(GetHeaderTag(siteName));

}

**Important:** Hungarian type notation should *not* be used for parameter names. Older Ignia code fails to comply with this standard and should be corrected when it is discovered.

## Private Fields

Private fields (sometimes referred to as global or member variables) should be prefixed with an underscore in order to differentiate them from local variables. As variables, they should use pascalCase; as such, a member variable might be called, for example, “\_email”.

## Hungarian Notation

As of .NET 1.0, Microsoft discourages use of Hungarian notation – and, in general, that should be followed in Ignia’s code. There are cases, however, where Hungarian notation (as originally intended[[2]](#footnote-2)) can be useful; most notably, it is helpful in avoiding collisions between identifiers that might otherwise use the same naming conventions. For example, a record set might be converted between several object types in a single scope (e.g., a DataSet, DataTable, DataAdapter, DataReader, etc); using Hungarian notation can help distinguish these (e.g., “dsUsers”, “dtUsers”, “daUsers”, “drUsers”). Similarly, using a prefix for declarative objects (such as form fields) can help avoid naming conflicts with properties; this is discussed more in “Form Fields” below.

## Form Fields

Declarative objects such as form fields may use Hungarian Notation type identifiers such as frm, fld, or btn. There are three benefits of this. First, it distinguishes fields from properties which may map to the same data (e.g., Email). Second, it distinguishes fields from local variables (e.g., email). Third, it maintains consistency with client side naming conventions (which tend to use camelCase).

# Indentation and Formatting

Ignia’s indentation structure differs slightly from standard C formatting conventions. This is a deliberate choice for readability although it is noted as inconvenient due to its inconsistency with sample code or code formatting tools (such as Visual Studios auto indentation).

## Indentation

Indentation should follow the page logic and natural nesting of code. Always use two *spaces* for indentation; if you are in the habit of using tabs, please set your editor to convert tabs to two spaces (this is supported by most popular editors, such as TextPad). The benefits of using two spaces include:

1. **Width:** It is short enough that it keeps most code within a viewable window.
2. **Consistency:** While tabs are interpreted differently by various clients, spaces are rendered consistently according to fixed-width font settings.

## Spacing

Ignia uses Microsoft’s Internal Coding Guidelines (Microsoft Corporation, 2005) for spacing; the following is excerpted directly from Brad Abram’s documentation:

* Do use a single space after a comma between function arguments.  
  Right: Console.In.Read(myChar, 0, 1);  
  Wrong: Console.In.Read(myChar,0,1);
* Do not use a space after the parenthesis and function arguments  
  Right: CreateFoo(myChar, 0, 1)  
  Wrong: CreateFoo( myChar, 0, 1 )
* Do not use spaces between a function name and parenthesis.  
  Right: CreateFoo()  
  Wrong: CreateFoo ()
* Do not use spaces inside brackets.  
  Right: x = dataArray[index];  
  Wrong: x = dataArray[ index ];
* Do use a single space before flow control statements  
  Right: while (x == y)  
  Wrong: while(x==y)
* Do use a single space before and after comparison operators  
  Right: if (x == y)  
  Wrong: if (x==y)

## Bracing

Opening braces “{“ should always appear on the *same* line as the statement that begins the block; closing braces should then appear on a new line, *indented to align with the content*. Braces should never be considered optional; this increases code readability and maintainability. The *only* exception to this is single statement blocks, in which case the statement should occur on the same line; if a single statement block is placed on a separate line then braces should be used.

## Example

public bool MethodName (String paramName) {

if (String.IsNullOrEmpty(paramName) return false;

if (bolCondition == true) {

intCounter++;

return true;

}

else {

return false;

}

}

# Comments

Comments should be used to describe intention, algorithmic overview, and/or logical flow (Microsoft Corporation, 2005). Another individual should be able to understand the structure, behavior and general logic of code based on the comments alone. Ignia uses a somewhat elaborate set of ASCII characters for identifying major comments (“flower boxing”); generally these are included in development templates, although it may be of value to add them to your editors clip library or associate them with hotkeys.

## Header Block

Every document containing code should begin with a consistent header block which includes the code title, author, project, purpose, and revision history. Generally, you should be using a standard template when establishing new code; in this case, the header block will be present and can simply be modified. Keeping this up-to-date is critical as it will help testers and future developers to track back bugs, especially for situations where dependencies are broken.

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

| [TITLE]

|

| Author [FName] [LName], Ignia LLC ([FName.LName]@ignia.com)

| Client [Client Name]

| Project [Project Name]

|

| Purpose [Purpose]

|

>==============================================================================

| Revisions Date Author Comments

| - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

| MM.DD.YY [FName] [LName] [Description of update]

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

## Member Block

Members (including methods and properties) should have a title and description. Often the friendly name is the same as the function name but with spaces and any abbreviations spelled out. While this is redundant it makes it easy for other developers (and you) to quickly scan the page for function calls.

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

| [METHOD|PROPERTY]: [FUNCTION FRIENDLY NAME]

>==============================================================================

| [Description].

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

Some developers choose to provide additional details with each function, following the basic template of the header block. These optionally include: purpose, edit history, parameter details, etc. If these are valuable to you or help clarify usage then please use them. This level of documentation, however, is not required and is often tedious and unnecessary to maintain for most code.

## Section Titles

If multiple major functional areas are represented inside a procedure then it’s useful to block them off using a comment header. This allows multiple lines of code that may be at the same level to be visually grouped together. For example, this type of title block may separate a database call or e-mail object.

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

| [SECTION TITLE]

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

## Inline Comments

Beyond this, simply use inline comments using the standard // syntax. Comments should be un-indented two spaces so that the comment itself lines up with the code; this helps with readability and also maintains indentation for code that is commented out. For instance:

public void FunctionName(String paramName) {

//Begin function

if (bolCondition) {

//intCounter++;

Response.Write(paramName);

}

}

In this sample, notice that “intCounter++;” maintains its proper indentation despite the preceding comment. Likewise, the text comment “Begin function” is clearly aligned and associated with the if-condition.

## Editorial Comments

There are a few special comment types which are often used. These are helpful for provided editorial feedback to reviewers, developers or future owners. These usually follow the general syntax:

//###[TYPE] [FML][MMDDYY]: [Description]

For instance,

//###BUG JJC101505: This code does not appear to work and needs to be reviewed.

### Popular Types

The following is a list of popular editorial comment types used at Ignia.

|  |  |
| --- | --- |
| BUG | Denotes a known issue or bug |
| TODO | Placeholder for code that is incomplete |
| HACK | Comment acknowledging an inelegant approach to a problem with an explanation of why this was used. |
| REM | Preceding temporarily commented out code explaining why it was removed and at what point it can be either permanently removed or uncommented. |
| EDIT | If a major change is made the edit may be recorded inline (as well as the revision history) to specifically mark a functional modification which may be of value to future developers editing or viewing the file. |

# Organization

Source files for class libraries should be organized in a consistent format to make it easy for developers to maintain applications. Folders should be modeled after namespaces; in turn, each file should represent (in contents and name) an individual type – although that type may include multiple internal classes. Within each file, members should be grouped by type (fields, constructors, properties, events, methods, etc) and alphabetized.

# Conclusion

While many of these rules may seem initially tedious they will quickly become a matter of habit and will aid both yourself and other developers in the creation, maintenance, review and updating of code.

# Works Cited

Microsoft Corporation. (2005, January 26). *Internal Coding Guidelines*. Retrieved 07 16, 2007, from Brad Abrams Blog: http://blogs.msdn.com/brada/articles/361363.aspx

Microsoft Corporation. (2005, 10 15). *Test*. Retrieved June 10, 2007, from Test: http://www.microsoft.com

1. Fields are also known as member variables and sometimes global variables. [↑](#footnote-ref-1)
2. See “I’m Hungary” from Joel on Software’s post “[Making Wrong Code Look Wrong](http://www.joelonsoftware.com/articles/Wrong.html)”. [↑](#footnote-ref-2)