# Coding Conventions

The following rules follow the standard rules for developing applications using .NET and C#.

## Naming Conventions

The following terms describe different ways to case identifiers.

***Pascal Casing***

The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters. For example:

BackColor

***Camel Casing***

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example:

backColor

***Uppercase***

All letters in the identifier are capitalized. For example:

IO

***Naming rules***

* Use Pascal casing for Class names.

public class HelloWorld

{

...

}

* Use Pascal casing for Method names

voidSayHello(string name)

{

...

}

* Use Camel casing for variables and method parameters

voidSayHello(string name)

{

stringfullMessage = "Hello " + name;

...

}

* All private member variables must be prefixed with underscore (\_) so that they can be identified from other local variables.

public class HelloWorld

{

privateint \_count;

}

* Use Meaningful, descriptive words to name variables. Do not use abbreviations.

string address (not addr)

int salary (not sal)

* Do not use variable names that resemble keywords.
* Prefix boolean variables, properties and methods with “is” or similar prefixes.

privatebool \_isFinished;

* Namespace names should follow the standard pattern.

***<product name>.<top level module>.<bottom level module>***

* Use appropriate prefix for the UI elements to identify them from the rest of the variables.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Control** | **Prefix** | | Label | lbl | | TextBox | txt | | DataGrid | dtg | | Grid | grd | | Button | btn | | DropDownList | ddl | | ListBox | lst | | Canvas | cvs | | DockPanel | dpn | | MediaElement | mde | | DataList | dtl | | Checkbox | chk | | CheckBoxList | cbl | | RadioButton | rdo | | Image | img | | TabControl | tct | | Slider | sld | | ProgressBar | pgb | |  |

* File name should match with class name. For the class HelloWorld, the file name should be HelloWorld.cs
* Use #region to group related pieces of code together. If you use proper grouping using #region, the page should like this when all definitions are collapsed.
* Keep private member variables, properties and methods in the top of the file and public members in the bottom.
* Compound words should be cased correctly (FxCopMicrosoft.NamingCA1702). Avoid creating compound words from terms which exist in the dictionary as discrete terms. Do not create a compound word such as 'StopWatch' or 'PopUp'. These terms are recognized in the dictionary and should be cased as 'Stopwatch' and 'Popup'.
* Identifiers should differ by more than case. (FxCopMicrosoft.NamingCA1708). Do not use names that require case sensitivity for uniqueness. Components must be fully usable from both case-sensitive and case-insensitive languages. Since case-insensitive languages cannot distinguish between two names within the same context that differ only by case, components must avoid this situation.
* Identifiers should have correct prefix (FxCopMicrosoft.NamingCA1715)

Prefix interface names with the letter 'I' to indicate that the type is an interface, as in IFormattable. Prefix generic type parameter names with the letter 'T' and provide them with descriptive names, as in Dictionary<TKey, TValue>, unless a single 'T' is completely self-explanatory, as in Collection<T>. Use Pascal casing for both interface and type parameter names. Use abbreviations sparingly. Do not use the underscore character. Do use similar names when defining a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the letter I prefixed on the interface name, as with Component and IComponent.

* Identifiers should have correct suffix (FxCopMicrosoft.NamingCA1710)

Types that extend certain base types have specified name suffixes. Types that extend Attribute, for example, should be suffixed in 'Attribute', as in ObsoleteAttribute. This rules checks types that extend several base types, including Attribute, Exception, EventArgs, IMembershipPermission, Stream, and others.

* Identifiers should not match keywords (FxCopMicrosoft.NamingCA1716). Identifiers, which conflict with reserved language keywords, should be avoided. Using a reserved keyword as an identifier makes it harder for consumers in other languages to use your API.Parameter names should not match member names (FxCopMicrosoft.NamingCA1719)

Parameter names should be distinct from member names.

public class HelloWorld

{

privateint \_count;

publicHelloWorld(int count)

{

\_count = count; //Should not change the parameter name to \_count

}

}

* Property names should not match get methods (FxCopMicrosoft.NamingCA1721). A Get method was found with the same name as a property. Get methods and properties should have names that clearly distinguish their function. See the design guidelines for information regarding choosing properties over methods.
* Type names should not match namespaces (FxCopMicrosoft.NamingCA1724). Identifiers, which conflict in whole or in part with namespace names, should be avoided. Names that describe the purpose or contents of a type are preferred.

## Coding Styles

* Avoid writing very long methods. A method should typically have 1~50 lines of code. If a method has more than 50 lines of code, you must consider re factoring into separate methods.
* Do not hardcode configuration data. Configuration data should be put into configuration file or database.
* Avoid using member variables. Declare local variables wherever necessary and pass it to other methods instead of sharing a member variable between methods. When sharing a member variable between methods, it will be difficult to track which method changed the value and when.
* Do not make the member variables public or protected. Keep them private and expose public/protected Properties.
* Do not have more than one class in a single file.
* Avoid public methods and properties, unless they really need to be accessed from outside the class. Use “internal” if they are accessed only within the same assembly.
* Declare variables as close as possible to where it is first used. Use one variable declaration per line.
* Use StringBuilder class instead of String when manipulating string objects in a loop. The String object works in weird way in .NET. Each time a string is appended, it is actually discarding the old string object and recreating a new object, which is a relatively expensive operations.
* Never do a 'catch exception and do nothing'. If hiding an exception, there is no way to notice the exception happened or not. Lot of developers uses this handy method to ignore non-significant errors. Best practices are to avoid exceptions by checking all the error conditions programmatically. In any case, catching an exception and doing nothing is not allowed. In the worst case, exceptions should be logged for later investigation and system should not be halted.
* When you arethrown an exception, use the throw statement without specifying the original exception. This way, the original call stack is preserved.
* Write your own custom exception classes if required in your application. Do not derive your custom exceptions from the base class SystemException. Instead, inherit from ApplicationException.
* A class should be declared as sealed if there is no further inheritance.
* Abstract types should not have constructors (FxCopMicrosoft.DesignCA1012). Public constructors for abstract types do not make sense because you cannot create instances of abstract types.
* Avoid empty interfaces (FxCopMicrosoft.DesignCA1040). Interfaces are meant to contain members that specify a set of behaviors. To mark or label a class, use an attribute instead of an empty interface.
* Avoid excessive parameters on generic types (FxCopMicrosoft.DesignCA1005). Avoid generic types with more than two type parameters as users have difficulties understanding what type parameters represent in types with long type parameter lists.
* Avoid out parameters (FxCopMicrosoft.DesignCA1021). Using out parameters might indicate a design flaw. Although there are legitimate times to use out parameters, their use frequently indicates a design that does not conform to the design guidelines for managed code.
* Declare types in namespaces (FxCopMicrosoft.DesignCA1050). A type should be defined inside a namespace to avoid duplication.
* Do not declare static members on generic types (FxCopMicrosoft.DesignCA1000). The syntax for calling static members on generic types is complex, as the type parameter has to be specified for each call.
* Do not declare protected members in sealed types (FxCopMicrosoft.DesignCA1047). Sealed types cannot be extended, and protected members are only useful if you can extend the declaring type. Sealed types should not declare protected members.
* Do not declare virtual members in sealed types (FxCopMicrosoft.DesignCA1048). Sealed types cannot be extended, and virtual members are only useful if you can extend the declaring type.
* Do not expose generic lists (FxCopMicrosoft.DesignCA1002). Do not expose List<T> in object models. Use Collection<T>, ReadOnlyCollection<T> or KeyedCollection<K,V> instead. List<T> is meant to be used from implementation, not in object model API. List<T> is optimized for performance at the cost of long-term versioning. For example, if you return List<T> to the client code, you will not ever be able to receive notifications when client code modifies the collection.
* Do not hide base class methods (FxCopMicrosoft.DesignCA1061). Defining a method in a derived class with the same name and parameters that are more weakly typed as one that is defined in the base class will obstruct access to the method defined in the base class.
* Do not overload operator equals on reference types (FxCopMicrosoft.DesignCA1046). Most reference types, including those that override System.Object.Equals, do not override the equality operator (==). Most languages provide a default implementation of this operator.
* Do not pass types by reference (FxCopMicrosoft.DesignCA1045). Although there are legitimate times to use reference parameters, such use frequently indicates a design that does not conform to the design guidelines for managed code.
* Enumerators should be strongly typed (FxCopMicrosoft.DesignCA1038). Types that implement IEnumerator should also provide a version of the Current property that returns a type other than Object. Implement the interface member explicitly and make the strongly typed version public.
* Exceptions should be public(FxCopMicrosoft.DesignCA1064). Exception classes should either be public, or have a non-generic public ancestor.
* Implement standard exception constructors (FxCopMicrosoft.DesignCA1032). Multiple constructors are required to correctly implement a custom exception. Missing constructors can make your exception unusable in certain scenarios. For example, the serialization constructor is required for handling exceptions in XML Web services.
* Lists are strongly typed (FxCopMicrosoft.DesignCA1039). IList implementations should also provide versions of the IList members that are strongly typed, namely they should specify types other than Object for method and property parameter and return types. Implement the interface members explicitly and make the strongly typed versions public. It is safe to ignore violations of this rule when you are implementing a new object-based collection, such as a linked list, where types based on your collection will determine what the strong type is. These types should expose strongly typed members.
* Override methods on comparable types (FxCopMicrosoft.DesignCA1036). Types that implement IComparable should redefine Equals and comparison operators to keep the meanings of less than, greater than, and equals consistent throughout the type.
* Replace repetitive arguments with params array (FxCopMicrosoft.DesignCA1025). Several instances of same-type arguments can be better implemented as a parameter array argument. Generally, if a member declares three or more arguments of the same type, consider using a parameter array.
* Static holder types should be sealed (FxCopMicrosoft.Design CA1052). Static holder types do not provide functionality that derived instances can extend. Inheriting from such a type indicates a flawed design.
* Use events where appropriate (FxCopMicrosoft.DesignCA1030). A method name suggestive of event functionality was encountered.
* Use generic event handler instances (FxCopMicrosoft.Design CA1007). Do not declare new delegates to be used as event handlers when targeting a version of the .NET Framework that supports generics. Use an instance EventHandler<T> instead.
* Use properties where appropriate (FxCopMicrosoft.DesignCA1024). Properties should be used instead of Get/Set methods in most situations. Methods are preferable to properties in the following situations: the operation is a conversion, is expensive or has an observable side-effect; the order of execution is important; calling the member twice in succession creates different results; a member is static but returns a mutable value; or the member returns an array.
* Do not hardcode locale specific strings (FxCopMicrosoft.GlobalizationCA1302). Do not use string literals for system items that have locale-specific designations. Special system locations should be retrieved using provided API such as GetFolderPath. See the System.Environment.SpecialFolder enumeration for more information.
* Set locale for data types (FxCopMicrosoft.GlobalizationCA1306). In most cases, Locale should be explicitly set to CultureInfo.InvariantCulture on DataSet and DataTable instances. Upon creation of a DataSet or DataTable instance, the Locale is set to the current culture. In most cases, the Locale should be set to CultureInfo.InvariantCulture to guarantee proper sorting behavior in all cultures.
* Specify CultureInfo (FxCopMicrosoft.GlobalizationCA1304). If an overload exists that takes a CultureInfo argument, it should always be called in favor of an overload that does not. The CultureInfo type contains culture-specific information required for performing numeric and string operations, such as casing, formatting, and string comparisons. In scenarios where conversion and parsing behavior should never change between cultures, specifyCultureInfo.InvariantCulture, otherwise, specify CultureInfo.CurrentCulture.
* Specify IFormatProvider (FxCopMicrosoft.GlobalizationCA1305). If an overload exists that takes an IFormatProvider argument, it should always be called in favour of an overload that does not. Some methods in the runtime convert a value to or from a string representation and take a string parameter that contains one or more characters, called format specifies, which indicate how the value is to be converted. If the meaning of the format specified varies by culture, a formatting object supplies the actual characters used in the string representation. In scenarios where sorting and comparison behavior should never change between cultures, specifyCultureInfo.InvariantCulture, otherwise, specify CultureInfo.CurrentCulture.
* Specify StringComparison (FxCopMicrosoft.GlobalizationCA1307). If an overload exists that takes a StringComparison argument, it should always be called in favour of an overload that does not.
* Use ordinal StringComparison (FxCopMicrosoft.GlobalizationCA1309). For non-linguistic comparisons, StringComparison.Ordinal or StringComparison.OrdinalIgnoreCase should be used instead of the linguistically-sensitive StringComparison.InvariantCulture.

## Comments

* Comments should be written in order to make code understandable and maintainable.
* Only write comments when necessary, do not write comments for every line of code and every variable declared.
* Use **//** or **///** for comments. Avoid using **/\* … \*/**
* Do not write comments if the code is easily understandable without comment. The drawback of having lot of comments is that if you change the code and forget to change the comment, it will lead to more confusion.
* Fewer lines of comments will make the code more elegant. But if the code is not clean/readable and there are less comments, that is worse.
* Writes clean, readable code such a way that it doesn't need any comments or very little to understand.
* Perform spelling check on comments and also make sure proper grammar and punctuation is used.

## Coding Conventions Specific for this Project

* All strings must be trimmed before inserting to the database. Empty strings (after trimmed) are inserted as null. (Framework handles this already, just for some special cases)
* All constants must be stored in UltraSub.Common.Constants class. Constant variables must be capitalized. For example: const string MAX\_QUERY = 5
* All configurable constants must be stored in config file App.config. For example: CultureLanguage
* All messages must be stored in UltraSub.Common.Messages class. Using Pascal case for message string variables. Prefix error message with the letter **E**, inform message with the letter **I** and warning message with the letter **W**. All messages must be defined in a message list. For example: const string E0001 = “Error is occurred!”
* All open source libraries must be installed via Nuget (if exist there).
* All UI resource must be stored in UltraSub.Presentations.Resources.
* Never use absolute path, use relative path instead. For example: “./Resource/Background.img” instead of “D:\Resource\Background.img”