General Coding Standards

Introduction

The guidelines detailed below form the basis for how code should be developed in BizTalk360 and other products should they wish. Developers, where ever possible, should implement the rules below into their day-to-day development, common sense trumps all after all.

This list should also change over time as development practices change and new technologies are introduced whilst older ones are removed.

Table of Contents

- 1. CSharp
  - 1.1. Naming
  - 1.2. Style
  - 1.3. Coding Practices
- 2. JavaScript
  - 2.1. Naming
  - 2.2. Style
  - 2.3. Coding Practices

1. CSharp

1.1. Naming

- Class names: Use pascal casing

```
// Don't
public class my_special_class {}
public class mySpecialClass {}

// Do
public class MySpecialClass {}
```

- Method names: Use pascal casing

```
// Don't
public void delete_alarm
public void deleteAlarm

//Do
public void DeleteAlarm
```

- Method arguments : Use camel casing

```
// Don't
public void UpdateAlarm(AlarmInfo AlarmInfo)

// Do
public void UpdateAlarm(AlarmInfo alarmInfo)
```

- Local Variable : Use camel casing

```
// Don't
public int SomeInt;
public int some_int;
//Do
public int someInt;
```

- Properties : Use pascal casing

```
// Don't
class TimePeriod
{
    private double _seconds;

    public double hoursSpent
    {
        get { return _seconds / 3600; }
        set { _seconds = value * 3600; }
    }
}

// Do
class TimePeriod
{
    private double _seconds;
    public double HoursSpent
    {
        get { return _seconds / 3600; }
        set { _seconds = value * 3600; }
    }
}
```

- Private & protected members: Prefix with "_"(underscore). Use camel casing for the rest.

```
// Don't
class TimePeriod
```

```
{
    private double secondsSpent;
    ...
}
// Do
class TimePeriod
{
    private double _secondsSpent;
    ...
}
```

- Interfaces: Prefix with "I"

```
// Don't
public interface Monitor
// Do
public interface IMonitor
```

- Constants: Use pascal casing

```
// Don't
public const int monthsInYear = 12;

// Do
public const int MonthsInYear = 12;
```

- Custom attribute classes: Suffix with attribute
- Custom exception classes: Suffix with exception
- Generic type arguments should always begin with T

```
class MyClass<T> {}
class MyClass<TInput, TOutput> {}
```

1.2. Style

- Do not use space for indentation. Use tabs and set to 2.
- Do not leave more than one empty lines between lines of code.
- Every file requires Copyright and Using Directives regions as per below.

```
#region Copyright © 2010 - 2016 BizTalk360.
// some copyright statements
#endregion
#region Using Directives
[ Blank Line ] ..
```

```
.Net using statements [ Blank Line ]
.. Third Party using statements [ Blank Line ]
.. BizTalk360 using statements [ Blank Line ]
.. Solution using statements [ Blank Line ]
#endregion
```

- Always use C# predefined types rather than the aliases in the System namespace.
  For example:
  object Not Object string Not String int Not Int32
- Comments
  - Classes, Methods & Properties: Use XML Comments. Use GhostDoc to help automatically build comments.
  - Code: Use the //comment style to apply comments on code. A comment should exist on the top of the line of group of lines of the code it is applied to . Multi line comments should look like this: // This is very big line for a comment and// therefore I wrap it to a second line like this. Comments should start with a space character and use sentence cse. A comment should finish with a full stop unless the text continues into a new line.
- Always open a curly brace in a new line
- Class definitionA class should declare its members in the following order
  - All private members (not methods)
  - All protected/internal members (not methods)
  - All constructors/destructors/IDisposable implementations
  - All public/internal Properties
  - All public/internal methods
  - All protected/private methods
- In a method, declare a local variable as close as possible to its first usage
- Use var for every local method variable type unless its absolutely necessary to use the actual type.

```
//Don't
public void DoSomething()
{
    int someInt = 0;
}

//Do
public void DoSomething()
{
    var someInt = 0;
}
```

- Use object and collection initializers

```
// Use
var myObj = new Alarm
{
```

```
    Name = "Test",
    Type = AlarmType.Daily,
    Emails = new List<string>
                {
                    "a@b.com",
                    "b@c.com"
                }
}

// Instead of:
var myObj = new Alarm();

myObj.Name = "Test";
myObj.Type = AlarmType.Daily;
myObj.Emails = new List<string>();
myObj.Emails.Add("a@b.com");
myObj.Emails.Add("b@c.com");
```

- Multi line blocks like If, for, foreach etc. Should not surround their code in curly braces if the code is just a simple single line statement. This is not a hard rule but is preferred as it produces cleaner looking code.

```
//Use
if(str == "blue") return 2;
// Instead of
if(str == "blue")
{
    return 2;
}
```

- Always use automatic properties over implemented properties when there is no embedded logic / processing needed for it.

```
// Use
public int Age { get; set;}
// Instead of
private int _time;

public int Time
{
    get { return _age; }
    set { _time = value }
}
```

- In case a property needs to be implemented the get and set statements should exist in a single line each if they contain only one statement

```
private string _name;

public string Name
{
    get { return _name; }
    set { _name = value.ToUpper(); }
}
```

- Class public properties when called in methods should NOT be prefixed with the this keyword.
- Incomplete code is indicated with a TODO comment.

1.3. Coding Practices

- All private members that are set by the constructor shouod be declared read only.
- Do not provide public member variables. Use properties instead.
- Use multiple return statements in a method to avoid nesting.
- Global variables are protected by locks or locking subroutines.
- Objects accessed by multiple thread are modified only through locks.
- Try to avoid methods with more than 100 lines.
- Throw as explicit
- Catch only exceptions that you explicitly handle.
- Do not ignore exceptions. If an exception is expected try to add code to that anticipates the exception before and try to prevent it.
- Use tertiary operator whenever possible and appropriate

```
//Don't
if(x > 0)
    return "foo";
else
    return "bar";
//Do
return (x>0) ? "foo" : "bar"
```

- When the two parts of a tertiary operator is long then the code should be formatted like this

```
var foo = (id != 0 && someVar > 5)
? GetDataFromDatabase (param1,param2,param3)
: GetDataFromMemory (param1,param2,param3);
```

- Use the null coalescing operator (??) whenever possible.

```
//Don't
if (x !=null )
  return x;
```

```
else
  return "foo";

//Do
return x ?? "foo"
```

- Use string.Empty instead of ""
- Never use a goto.
- Remove not used using statements from each code file.

2. JavaScript

2.1. Naming

- Constructor function should always start with a capital letter

```
//Don't
var operations = new operations();
//Do
var operations = new Operations();
```

- Method/Function should always start with a small letter

```
//Don't
function GetQueryList() { ... }
//Do
function getQueryList() { ... }
```

- Use camel case for identifier names (variables and functions)

```
pageTitle = "Operations";
```

- Write global variables in **UPPERCASE**
- Write constants in **UPPERCASE**
- Use **lower case** for file names
- use underscore in the begining of the private identifier

```
var Person = function() {
    var _age  = 0,
        _name = "John Doe";
};
```

2.2. Style

- Use double quotes instead of single quotes

```
//Don't
name = 'O''reilly media';
//Do
name = "O'reilly media";
```

- Always use blocks with structured statements such as *if* and *while* because it is less error prone

```
//Don't
if(a)
    b();
//Do
if(a){
    b();
}
```

```
// ================================
//            EXAMPLE
// ================================
if(a)
    b();
//can become
if(a)
    b();
    c();
//which is an error very difficult to spot. It looks like
if(a){
    b();
    c();
}
//but it means
if(a){
    b();
}
c();
```

- Use only meaningful comments

```
// Don't
var i = 0; //Set i to zero
// Do
var i = 0;
```

- Prefer line comments, use block comments only for formal documentation and for commenting out
- Make the code self-illuminating. Try to eliminate the need for comments.

- Declare all your variables at the begining of each function. The convention that variables should be declared at their first use is not suitable for JavaScript since JavaScript has function scope not block scope.
- Don't assign variables inside if condition

```
if(a = b) { ... }
is probably intended to be:
if(a === b) { ... }
```

- Put a semicolon at the end of every simple statement

2.3. Coding Practices

- Don't use global variables. There are three ways to define global variables. Avoid the usage of all the three except. You should use single global variable to contain your application or library.

```
// The first is to place a var statement outside of any function
var foo = value;
// The second is to add a property directly to the global object
window.foo = value;
// The third one is to use a variable without declaring it. This is called implied
global
foo = value;
```

- JavaScript has a mechanism to correct faulty programs by automatically inserting semicolons. Do not depend on this. For example the below code will return *undefined* no matter what the value of the status might be.

```
//Don't
return
{
    status: true
};
//Do
return {
    status: true
}
```

- Don't use typeof for checking objectness of a variable. since *typeof null* returns *object*

```
// Don't
typeof my_value === null
// Do
if(my_value && typeof my_value === "object") {
    // your code
}
```

- Use === and !== for equality checks

```
// Don't
if(my_value == 5) { ... }
// Do
if(my_value === 5){ ... }
```

- When using Durandal declare each identifier of the module separately in both singleton and transient module type

```
define([
    'durandal/system'
],
function(system){
    var model = ko.observable();
    var name = ko.observable();
    .
    .
    .
    var activate = function(){
        .
        .
        .
    }

});
```

```
define(function(require){
    var system = require('durandal/system');

    var AzureServices = function() {
        this.model = ko.observable();
        this.name = ko.observable();
    }

    AzureServices.prototype.activate = function(){
        var self = this;
        // code here
    }

    return AzureServices;
})
```