Coding Conventions

Purpose of Having Coding Standards

- A coding standard gives a uniform appearance to the codes written by different engineers.
- It improves readability, and maintainability of the code and it reduces complexity also.
- It helps in code reuse and helps to detect error easily.
- It promotes sound programming practices and increases efficiency of the programmers.l

General Instructions

- Always keep SECURITY in mind while coding. There should not be any type of data leakage.
- Always keep the static data in a separate file. Never hardcode a value in the code.
- Use meaningful Identifier names.
- Before starting with a task/project please analyse it thoroughly and get you queries cleared.
- Comment the code wisely.
- Always use preferred architecture for code.
- Always try to make the code component oriented.
- Make the components generic, so that they can be used in multiple projects.
- Try to avoid use of globals where possible.
- Always use exception handling properly and don’t forget to log the errors.
- Avoid using an identifier for multiple purposes.
- Try to keep the functions short and to the point (Max 15 lines).
- Make single responsibility functions.
- Don’t duplicate the code EVER.
- Only add a single class in one .cs file.
- Always try to use dependency injection.
- Don’t use GOTO Statements.
- **Add Unit test project with every project and its developer's duty to write cases and test your own code before sending it to QA/PM or Client.**
- Take care of proper spacing and indentation of code.
- Always use validations for data coming from untrusted sources.
- Write everything code-related in english.
- Prefer to write efficient code e.g use switch statements instead of multiple if conditions or use foreach loop instead of for loop.
- Always create normalized database structure.
- Try to use SCSS/SASS instead of CSS.
- All the static files css/js/images/fonts should be in a single parent folder and then their respective folders for file types.
- Avoid in-line or in-page styling/scripts and use stylesheets and js files instead.
- Use minified js/css files for increased performance.
- Only load necessary css/js for the page.
- Error messages should be meaningful.
- Always use html tags in lower case.
- Always try to include alt attributes for image tags.
- Every web page should have an appropriate title.
- Css classes names should be in lowercase.
- When working on a web app, please try to add PWA capabilities in it.

C# Coding Standards and Naming Conventions

Do use PascalCasing for class names and method names .Use nouns or noun phrases to name a class. For methods use verbs or verb phrases.

```
public class ClientActivity
{
   public void ClearStatistics()
   {
       //...
   }
   public void CalculateStatistics()
   {
       //...
   }
}
```

Do use PascalCasing for property names. Use nouns or noun phrases to name a property. Avoid using hugarian notations in property names.

```
public string EmployeeName { get; set; }
```

Do use Pascal case for namespaces.

```
Microsoft.Media
```

Do use the company name followed by the technology name and optionally the feature and design when naming the namespaces as follows.

```
CompanyName.TechnologyName[.Feature][.Design]
```

A nested namespace should have a dependency on types in the containing namespace. For example, the classes in the System.Web.UI.Design depend on the classes in System.Web.UI. However, the classes in System.Web.UI do not depend on the classes in System.Web.UI.Design.

Do use camelCasing for local variables and method arguments.

```
public class UserLog
{
   public void Add(LogEvent logEvent)
   {
       int itemCount = logEvent.Items.Count;
   }
}
```

Don't use Hungarian notation or any other type identification in identifiers.

```
// Avoid
int iCounter;
string strName;

// Correct
int counter;
string name;
```

Don't use Screaming Caps for constants or readonly variables.

```
// Avoid
public static const string SHIPPINGTYPE = "DropShip";

// Correct
public static const string ShippingType = "DropShip";
```

Don't use Underscores in identifiers. Exception: you can prefix private static variables.

```
// Avoid
public DateTime client_Appointment;

// Correct
public DateTime clientAppointment;

// Exception
private DateTime _registrationDate;
```

Avoid using Abbreviations. Exceptions: abbreviations commonly used as names, such as Id, Xml, Ftp, Uri.

```
// Avoid
UserGroup usrGrp;
Assignment empAssignment;

// Correct
UserGroup userGroup;
Assignment employeeAssignment;

// Exceptions
CustomerId customerId;
XmlDocument xmlDocument;
FtpHelper ftpHelper;
UriPart uriPart;
```

Do use PascalCasing for abbreviations 3 characters or more (2 chars are both uppercase).

```
HtmlHelper htmlHelper;
FtpTransfer ftpTransfer;
UIControl uiControl;
```

Do use predefined type names instead of system type names like Int16, Single, UInt64, etc.

```
// Correct
string firstName;
int lastIndex;
bool isSaved;

// Avoid
String firstName;
Int32 lastIndex;
Boolean isSaved;
```

Do use implicit type var for local variable declarations. Exception: primitive types (int, string, double, etc) use predefined names.

```
// Correct
var stream = File.Create(path);
var customers = new Dictionary();

// Exceptions
int index = 100;
string timeSheet;
bool isCompleted;
```

Do use noun or noun phrases to name a class.

```
public class Employee
{
}
public class BusinessLocation
{
}
public class DocumentCollection
{
}
```

Do prefix interfaces with the letter **I**. Interface names are nouns (phrases) or adjectives.

```
public interface IShape
{
}
public interface IShapeCollection
{
}
public interface IGroupable
{
}
```

Do name source files according to their main classes. Exception: file names with partial classes reflect their source or purpose, e.g. designer, generated, etc.

```
// Located in Task.cs
public partial class Task
{
   //...
}
// Located in Task.generated.cs
public partial class Task
{
   //...
}
```

Do use function overloading to implement an operation which achieves the same purpose but in a different way and it needs different parameters.

```
// Correct
public User FindUser(int id)

public User FindUser(string username)

// Avoid
public User FindUserById(int id)

public User FindUserByUserName(string username)
```

Do organize namespaces with a clearly defined structure.

```
// Examples
namespace Company.Product.Module.SubModule
namespace Product.Module.Component
namespace Product.Layer.Module.Group
```

Do vertically align curly brackets.

```
// Correct
class Program
{
   static void Main(string[] args)
   {
   }
}
```

Do declare all member variables at the top of a class, with static variables at the very top.

```
// Correct
public class Account
{
   public static string BankName;
   public static decimal Reserves;

   public string Number {get; set;}
   public DateTime DateOpened {get; set;}
   public DateTime DateClosed {get; set;}
   public decimal Balance {get; set;}

   // Constructor
   public Account()
   {
       // ...
   }
}
```

Do use singular names for enums in PascalCase. Exception: bit field enums.

```
// Correct
public enum Color
{
   Red,
   Green
}

// Exception
[Flags]
public enum Dockings
{
   None = 0,
   Top = 1,
   Right = 2,
   Bottom = 4,
   Left = 8
}
```

Don't explicitly specify a type of an enum or values of enums (except bit fields).

```
// Avoid
public enum Direction : long
{
   North = 1,
   East = 2,
   South = 3,
   West = 4
}

// Correct
public enum Direction
{
   North,
   East,
   South,
   West
}
```

Don't suffix enum names with Enum.

```
// Avoid
public enum CoinEnum
{
   Penny,
   Nickel
}
// Correct
public enum Coin
{
   Penny,
   Nickel,
   Dollar
}
```

Do use appropriate suffixes while naming classes of type Attribute, Filter, Controller, EventArgs etc.

```
// Correct
public class AuthorizationAttribute : Attribute
public class ValidateXSSFilter : ActionFilter
public class AccountController : Controller
public class MouseEventArgs : EventArgs
```

Do use appropriate suffixes while naming methods of type EventHandler etc.

```
// Correct
public void MouseEventHandler(object sender, MouseEventArgs e)
```

Do use inbuilt tools provided by the platform instead of making custom statements e.g string.IsNullOrEmpty() or StringBuilder.

```
// Correct
if(string.IsNullOrEmpty(myString));

// Avoid
if(myString == "");
```

Do use delegates when you want to represent or refer to one or more functions or to define call-back methods and implement event handling.

Do use the using keyword when working with disposable types. It automatically disposes the object when program flow leaves the scope.

```
using(var conn = new SqlConnection(connectionString))
{
    // use the connection and the stream
    using (var dr = cmd.ExecuteReader())
    {
     //
    }
}
```

Do use abstract classes when you want to have certain concrete methods and some other methods that the derived classes should implement. By contrast, if you use interfaces, you would need to implement all the methods in the class that extends the interface. An abstract class is a good choice if you have plans for future expansion – i.e. if a future expansion is likely in the class hierarchy. If you would like to provide support for future expansion when using interfaces, you'll need to extend the interface and create a new one.

JS Coding Standards and Naming Conventions

Do use camelCase for variables.

Do use PascalCase for class names.

Do use camelCase for functions.

Do use lowercase for js file names.

Do use === instead of == when comparing.

```
// Correct
if (val === 2)

// Avoid
if (val == 2)
```

Do use `let` instead of `var`.

```
// Correct
let myVar = 10;

// Avoid
var myVar = 10;
```

Do use `const` for the constants.

```
// Correct
const VAT_PERCENT = 20;

// Avoid
let VAT_PERCENT = 20;
```

Do use semicolons (;)

```
// Correct
const vatPercent = 20;
let amount = 10;
return addVat(amount, vatPercent);

// Avoid
const vatPercent = 20;
let amount = 10
return addVat(amount, vatPercent)
```

Do use `const` for the constants.

```
// Correct
const VAT_PERCENT = 20;

// Avoid
let VAT_PERCENT = 20;
```

Do Use template literals when contacting strings (Not in the case when you want to support IE 11 or lower).

```
// Correct
let fullName = `${firstName} ${lastName}`;

// Avoid
let fullName = firstName + " " + lastName;
```

Do Use ES6 arrow functions where possible.

```
// Correct
const multiply = (a, b) => { return a * b};

// Avoid
var multiply = function(a, b) {
 return a* b;
};
```

Do Always use curly braces around control structures (i.e. if, else, for, do, while, as well as any others).

```
// Correct
if (valid) {
  doSomething();
}

// Avoid
if (valid)
  doSomething();
```

Also, make sure in JavaScript that the curly brace starts on the same line with space in between.

```
// Correct
if (myNumber === 0) {
   doSomething();
}

// Avoid
if (myNumber === 0)
{
   doSomething();
}
```

Do use default parameters where possible.
In JavaScript, if you don't pass in a value into a parameter when calling a function it will be undefined.

```
// Correct
myFunction(a = 0, b = 0) {
  return a + b;
}

// Avoid
myFunction(a, b) {
 return a + b;
}
```

Do Try and reduce nesting.

```
// Correct
if (myNumber <= 0) {
  return error;
}
if (!hasDiscountAlready) {
   return addDiscountPercent(0);
}
if (myNumber > 100) {
   return addDiscountPercent(10);
}
if (myNumber > 50) {
   return addDiscountPercent(5);
}
return addDiscountPercent(1);
```

```
// Avoid
if (myNumber > 0) {
 if (myNumber > 100) {
       if (!hasDiscountAlready) {
           return addDiscountPercent(0);
       } else {
           return addDiscountPercent(10);
       }
 } else if (myNumber > 50) {
   if (!hasDiscountAlready) {
      return addDiscountPercent(5);
   }
 } else {
   if (!hasDiscountAlready) {
     return addDiscountPercent(0);
   } else {
     return addDiscountPercent(1);
   }
 }
} else {
    error();
}
```

Do use `break` and `default` in switch statements.

```
// Correct
switch (myNumber)
{
 case 10:
   addDiscountPercent(0);
   break;
 case 20:
   addDiscountPercent(2);
   break;
 case 30:
   addDiscountPercent(3);
   break;
 default:
   addDiscountPercent(0);
   break;
}

// Avoid
switch (myNumber)
{
 case 10:
  addDiscountPercent(0);
 case 20:
  addDiscountPercent(2);
 case 30:
  addDiscountPercent(3);
}
```

Do use named exports instead of default exports.

```
// Correct
export class MyClass {

// Avoid
export default class MyClass {
```

Do use named exports instead of default exports.

```
// Correct
export class MyClass {

// Avoid
export default class MyClass {
```

Don't use wildcard imports.

```
// Avoid
import * as Foo from './Foo';

// Correct
import Foo from './Foo';
```

Do use shortcuts for booleans.

```
// Correct
if (isValid)
if (!isValid)

// Avoid
if (isValid === true)
if (isValid === false)
```

Avoid unneeded ternary statements.

```
// Avoid
const boo = a ? a : b;

// Correct
const boo = a || b;
```