C# Coding Standards and Naming Conventions

Below are our C# coding standards, naming conventions, and best practices.
Use these in your own projects and/or adjust these to your own needs.

---

do use PascalCasing for class names and method names.

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

Why: consistent with the Microsoft's .NET Framework and easy to read.

---

do use camelCasing for method arguments and local variables.

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

```
    }
}
```

Why: consistent with the Microsoft's .NET Framework and easy to read.

---

do not use Hungarian notation or any other type identification in identifiers

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

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

Why: consistent with the Microsoft's .NET Framework and Visual Studio IDE makes determining types very easy (via tooltips). In general you want to avoid type indicators in any identifier.

---

do not use Screaming Caps for constants or readonly variables

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

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

Why: consistent with the Microsoft's .NET Framework. Caps grap too much attention.

---

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

```
// Correct
UserGroup userGroup;
Assignment employeeAssignment;

// Avoid
UserGroup usrGrp;
Assignment empAssignment;

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

Why: consistent with the Microsoft's .NET Framework and prevents inconsistent abbreviations.

---

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

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

Why: consistent with the Microsoft's .NET Framework. Caps would grap visually too much attention.

---

do not use Underscores in identifiers. Exception: you can prefix private static variables with an underscore.

```
// Correct
public DateTime clientAppointment;
public TimeSpan timeLeft;
```

```
4.
5.  // Avoid
6.  public DateTime client_Appointment;
7.  public TimeSpan time_Left;
8.
9.  // Exception
10. private DateTime _registrationDate;
```

Why: consistent with the Microsoft's .NET Framework and makes code more natural to read (without 'slur'). Also avoids underline stress (inability to see underline).

---

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

```
1.  // Correct
2.  string firstName;
3.  int lastIndex;
4.  bool isSaved;
5.
6.  // Avoid
7.  String firstName;
8.  Int32 lastIndex;
9.  Boolean isSaved;
```

Why: consistent with the Microsoft's .NET Framework and makes code more natural to read.

---

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

```
1.  var stream = File.Create(path);
2.  var customers = new Dictionary();
3.
4.  // Exceptions
5.  int index = 100;
```

```
string timeSheet;
bool isCompleted;
```

Why: removes clutter, particularly with complex generic types. Type is easily detected with Visual Studio tooltips.

---

do use noun or noun phrases to name a class.

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

Why: consistent with the Microsoft's .NET Framework and easy to remember.

---

do prefix interfaces with the letter `I`. Interface names are noun (phrases) or adjectives.

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

Why: consistent with the Microsoft's .NET Framework.

---

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
{
    //...
}
```

Why: consistent with the Microsoft practices. Files are alphabetically sorted and partial classes remain adjacent.

---

do organize namespaces with a clearly defined structure

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

Why: consistent with the Microsoft's .NET Framework. Maintains good organization of your code base.

---

do vertically align curly brackets.

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

Why: Microsoft has a different standard, but developers have overwhelmingly preferred vertically aligned brackets.

---

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()
    {
        // ...
    }
}
```

Why: generally accepted practice that prevents the need to hunt for variable declarations.

---

do use singular names for enums. Exception: bit field enums.

```
// Correct
public enum Color
{
    Red,
    Green,
    Blue,
    Yellow,
    Magenta,
    Cyan
}

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

Why: consistent with the Microsoft's .NET Framework and makes the code more natural to read. Plural flags because enum can hold multiple values (using bitwise 'OR').

---

do not explicitly specify a type of an enum or values of enums (except bit fields)

```
// Don't
public enum Direction : long
{
    North = 1,
    East = 2,
    South = 3,
    West = 4
}

```

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

Why: can create confusion when relying on actual types and values.

---

do not suffix enum names with Enum

```
// Don't
public enum CoinEnum
{
    Penny,
    Nickel,
    Dime,
    Quarter,
    Dollar
}

// Correct
public enum Coin
{
    Penny,
    Nickel,
    Dime,
    Quarter,
    Dollar
}
```

Why: consistent with the Microsoft's .NET Framework and consistent with prior rule of no type indicators in identifiers.