# C # Codificación de Normas y Convenciones de nomenclatura

# A continuación se presentan los estándares de codificación C #, convenciones de nombres, y las mejores prácticas.

# Utilícelos en sus proyectos y/o ajustelos a sus propias necesidades.

# Do use PascalCasing for class names and method names.

# (No utilizar PascalCasing de nombres de clases y nombres de los métodos.)

1. public class ClientActivity
2. {
3. public void ClearStatistics()
4. {
5. //...
6. }
7. public void CalculateStatistics()
8. {
9. //...
10. }
11. }

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

douse **camelCasing** for method arguments and local variables. **(NO utilizar camelCasing de argumentos del método y las variables locales.**

1. public class UserLog
2. {
3. public void Add(LogEvent logEvent)
4. {
5. int itemCount = logEvent.Items.Count;
6. // ...
7. }
8. }

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

do notuse **Hungarian** notation or any other type identification in identifiers

1. // Correct
2. int counter;
3. string name;
5. // Avoid
6. int iCounter;
7. 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 notuse **Screaming Caps** for constants or readonly variables

1. // Correct
2. public static const string ShippingType = "DropShip";
4. // Avoid
5. public static const string SHIPPINGTYPE = "DropShip";

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

avoidusing **Abbreviations**. Exceptions: abbreviations commonly used as names,   
                 such as **Id, Xml, Ftp, Uri**

1. // Correct
2. UserGroup userGroup;
3. Assignment employeeAssignment;
5. // Avoid
6. UserGroup usrGrp;
7. Assignment empAssignment;
9. // Exceptions
10. CustomerId customerId;
11. XmlDocument xmlDocument;
12. FtpHelper ftpHelper;
13. UriPart uriPart;

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

douse **PascalCasing** for abbreviations 3 characters or more (2 chars are both uppercase)

1. HtmlHelper htmlHelper;
2. FtpTransfer ftpTransfer;
3. UIControl uiControl;

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

do notuse **Underscores** in identifiers. Exception: you can prefix private static variables   
                    with an underscore.

1. // Correct
2. public DateTime clientAppointment;
3. public TimeSpan timeLeft;
5. // Avoid
6. public DateTime client\_Appointment;
7. public TimeSpan time\_Left;
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).

douse **predefined type names** instead of system type names like Int16, Single, UInt64, etc

1. // Correct
2. string firstName;
3. int lastIndex;
4. bool isSaved;
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. 

douse 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();
4. // Exceptions
5. int index = 100;
6. string timeSheet;
7. bool isCompleted;

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

douse noun or noun phrases to name a class.

1. public class Employee
2. {
3. }
4. public class BusinessLocation
5. {
6. }
7. public class DocumentCollection
8. {
9. }

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

doprefix interfaces with the letter **I**.  Interface names are noun (phrases) or adjectives.

1. public interface IShape
2. {
3. }
4. public interface IShapeCollection
5. {
6. }
7. public interface IGroupable
8. {
9. }

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

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

1. // Located in Task.cs
2. public partial class Task
3. {
4. //...
5. }
6. // Located in Task.generated.cs
7. public partial class Task
8. {
9. //...
10. }

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

doorganize namespaces with a clearly defined structure

1. // Examples
2. namespace Company.Product.Module.SubModule
3. namespace Product.Module.Component
4. namespace Product.Layer.Module.Group

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

dovertically align curly brackets.

1. // Correct
2. class Program
3. {
4. static void Main(string[] args)
5. {
6. }
7. }

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

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

1. // Correct
2. public class Account
3. {
4. public static string BankName;
5. public static decimal Reserves;
7. public string Number {get; set;}
8. public DateTime DateOpened {get; set;}
9. public DateTime DateClosed {get; set;}
10. public decimal Balance {get; set;}
12. // Constructor
13. public Account()
14. {
15. // ...
16. }
17. }

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

douse singular names for enums. Exception: bit field enums.

1. // Correct
2. public enum Color
3. {
4. Red,
5. Green,
6. Blue,
7. Yellow,
8. Magenta,
9. Cyan
10. }
12. // Exception
13. [Flags]
14. public enum Dockings
15. {
16. None = 0,
17. Top = 1,
18. Right = 2,
19. Bottom = 4,
20. Left = 8
21. }

**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 notexplicitly specify a type of an enum or values of enums (except bit fields)

1. // Don't
2. public enum Direction : long
3. {
4. North = 1,
5. East = 2,
6. South = 3,
7. West = 4
8. }
10. // Correct
11. public enum Direction
12. {
13. North,
14. East,
15. South,
16. West
17. }

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

do notsuffix enum names with Enum

1. // Don't
2. public enum CoinEnum
3. {
4. Penny,
5. Nickel,
6. Dime,
7. Quarter,
8. Dollar
9. }
11. // Correct
12. public enum Coin
13. {
14. Penny,
15. Nickel,
16. Dime,
17. Quarter,
18. Dollar
19. }

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

**http://www.nebaris.com/post/238/escribir-mejor-codigo-en-c-sharp**

**http://www.dofactory.com/reference/csharp-coding-standards**