Coding Conventions

(Adopted by Teenovators)

1.Naming Conventions :

i. **Pascal Casing :** First character of all words are upper case and other character are lower case. We will use pascal casing in the following cases :

- Class names.

```
Public class HelloWord()
     {
     ...................//class body
    }
```

- Function names.

```
Int SetProcessID(int processID, boolean isCamelCase)
{
................//function body
}
```

- Events.

```
buttonName.ValueChange("New Value");
```

- Namespace or package

```
import System.Drawing ;
```

- Property

```
buttonName.Color(BlackColor);
```

- Enumeration type

```
public enum ErrorLevel
{
FatalError, NoClassFoundException, NoDefinationFound
}
```

ii. **Camel Casing :** First character of all words, except the first word are upper case other characters are lower case. We will use camel casing in the following cases :

- Variable names

```
string firstName;
```

- Parameters of methods

```
int angleCounterClockWise;
```

- For an interface 'I' with camel case name of interface.

```
Ientity;
```

iii. **Meaningful names :** Use meaningful, descriptive variable name and don't use abbreviation.

```
String address ;                    // correct one
int angleCounterClockWise ;         // correct one
string add;                         // don't use abbreviation
```

iv. Don't use single character like i, j, k rather than use index etc. single character can be used as iterators in loops.

v. **UnderScore for variable name** :

- Don't use underscore(_) for local variables.

- Use underscore(_) for member variable, so distinction between local and member variable will be more clear.

vi. File name should match with the class name.

vii. Use capital letters for constant variables.

viii. Boolean variables should contain "is" prefix.

```
Boolean isValueChanged;
boolean _isDeveloped;
```

2. Indentation and code formatting :

i. Use tab for indentation rather than space and set tab is equal to 4 spaces.

ii. Curly braces ({}) should be in the same level as the code outside the braces.

iii. The curly braces should be on a separate line and not in the same line as if, for etc.

iv. Keep private member variables, properties and methods in the top of the file and public members in the bottom.

v. 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.

vi. Length of a line should be less than 50 characters. In case of longer line wrap the lines in different parts

Wrapping Of Line :

- Break after a comma, after an operator and align the new line with the beginning of the expression at the same level on the previous line.

```
Bad practice :
function(longExpression1,longExpression2,longExpression3,longExression);

Good practice :
function(longExpression1,
            longExpression2,
            longExpression3,
             longExpression4);
```

- Give high level breaks to lower level breaks

```
longName1 = longName2 * (longName3 + longName4
                                    - longName5) + 4 * longname6;

longName1 = longName2 * ( longName3 + longName4 – longName5 )+                    4 *
longname6;
```

vii.Blank Lines :

- Between two logical sections two improve readability.
- Between each method inside a class file.
- Between local variable and first line of method.
- Two blank line between two different sections.
- Two blank line between class and interface definition.

viii.Blank Space :

1. Use a single space before and after each operator and brackets, except in unary operators

```
Bad practice:
if(showResult==true) // put a spcae before and after ==
{
        for(int     i= 0;i<10;i++) // put a space before =, before and
            //    after <
        {
                //
        }
}

Good practice:
if ( showResult == true )
{
        for ( int i = 0; i < 10; i++ )
        {
                //
        }
}
```

2. A keyword followed by a parenthesis should be separated by a space.

```
Example:
while (true)
{
        ...
}
```

NOTE : A blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

3. A blank space should appear after commas in argument lists.

```
Int SetProcessID(int processID, boolean isCamelCase, string nameOfOperator)
{
................//function body
}
```

4. Casts should be followed by a blank.

```
myMethod((byte) aNum, (Object) x);
```

3.Comments :

1. Implementation Comment :

i. **Block comment :** Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments should be used at the beginning of each file and before each method.A block comment should be preceded by a blank line to set it apart from the rest of the code. Block comments have an asterisk "*" at the beginning of each line except the first.

```
/*
 * Here is a block comment with some very special
 * formatting that I want indent(1) to ignore.
 *
 * one
 * two
 * three
*/
```

ii. **Single-Line Comment :** Short comments can appear on a single line indented to the level of the code that follows. A single-line comment should be preceded by a blank line.

```
if (condition)
{
        /* Handle the condition. */
        ...
}
```

iii. **Trailing Comment :** Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements.

```
if (a == 2)
{
        return TRUE;                                    /* special case */
}
else
{
        return isprime(a);                      /* works only for odd a */
}
```

iv. **End-Of-Block Comment :** The // comment delimiter begins a comment that continues to the newline, which is used in end of a block.

```
int func1( int para1, int para2)
```

```
{
        while()
        {
                ...
        }// end of while
}//end of function1
```

4.Statements :

i. if-else statements :

```
if (condition)
{
        statements;
}
else if (condition)
{
        statements;
}
else if (condition)
{
        statements;
}
```

ii. for statements :

```
for (initialization; condition; update)
{
        statements;
}
```

iii. do-while Statements :

```
do
{
        statements;
} while (condition);
```

iv. switch statements :

```
switch (condition)
{
        case ABC:
                statements;
                /* falls through */
        case DEF:
                statements;
                break;
        case XYZ:
                statements;
                break;
        default:
                statements;
                break;
        }
```

v. try-catch Statements :

```
        try
        {
                statements;
        }       catch (ExceptionClass e)
                {
                        statements;
                }
```

5.Good programming practices :

i. Methods or Functions :

- ➔ Avoid writing long functions. It should typically of 1-25 line, if more than that then make sub functions.

- ➔ Method names should be very descriptive which will completely tell about what it does.

```
Bad Practice :
void Phone( int n )
{
        _phoneNumber = n;
}

Good practice :
void SetPhoneNumber( int number )
{
        _phoneNumber = number;
}
```

- ➔ A function should do only one job, do not combine two functions.

```
Bad Practice :
void Mail( string mail, string address )
{
        ...//send a mail
        ...//save address
}

Good practice
void SendMail( string mail )
{
        ...//send a mail
}
void SaveAddress( string address )
{
        ...//save address
}
```

- ➔ Avoid passing too many parameters to a method. If you have more than 4-5 parameters, it is a good candidate to define a class or structure and then pass that structure or class.

ii. Always look for unexpected values.For example, if you are using a parameter with 2 possible values, never assume that if one is not matching then the only possibility is the other value.

Bad Pracise :

```
    if ( memberType == eMember.Registered())
    {
            //member is the registered user
    }
    else
    {
            //if member is not registered than execute this block
    }
```

Good Practice :

```
    if ( memberType == eMember.Registered())
    {
            //member is the registered user
    }
    else if ( memberType == eMember.Guest() )
    {
            //if member is not registered than check for guest
    }
    else
    {
            // otherwise like if user not exists handle them here
    }
```

iii. Do not hardcode numbers in the code, rather than that use constants. These constants should be initialized by config file or database so we can vary them.

iv. Before comparing two string convert them into lower case or in upper case;

Bad practice :

```
    if ( name == "gaurav" )
    {
            bounus++;
    }
```

Good practice :

```
    if ( name.ToLower() == "gaurav" )
    {
            bounus++;
    }
```

Bad practice :

```
    if ( name == "" )
    {
            // name string is empty
    }
```

Good practice :

```
    if ( name == String.Empty )
    {
            // name string is empty

    }
```

v. Do not make the member variable public or protected, make them private and expose them to public methods.

vi. Use more descriptive error messages like " error in updating database with login information ". Other than error message it should contain solution of the error like " error in updating database with login information, Please check your login information "

vii. Declare variables as close as possible to where it is first used. Use one variable declaration per line.

viii. Always catch only the specific exception, not generic exception.

```
Bad practice :
    void ReadFromFile ( string fileName )
    {
            try
            {
                    // read from file.
            }       catch (Exception ex)
                    {
                            // all type of exception
                            return "";
                    }
            }

Good practice :
    void ReadFromFile ( string fileName )
    {
            try
            {
                    // read from file.
            }
            catch (FileIOException ex)
            {
                    // log error.
                    //  re-throw exception depending on your case.
                    throw;
            }
    }
```

Enjoy Coding,
Teenovators